Brewtils Documentation

Release 3.0.1

Logan Asher Jones

Dec 15, 2020

Contents

Contents:

CHAPTER 1

Brewtils

Brewtils is the Python library for interfacing with Beergarden systems. If you are planning on writing beer-garden plugins, this is the correct library for you. In addition to writing plugins, it provides simple ways to query the API and is officially supported by the beer-garden team.

1.1 Features

Brewtils helps you interact with beer-garden.

- Easy way to create beer-garden plugins
- Full support of the entire Beer-Garden API
- Officially supported by the beer-garden team

1.2 Installation

To install brewtils, run this command in your terminal:

```
$ pip install brewtils
```

Or add it to your `requirements.txt`

```
$ cat brewtils >> requirements.txt
$ pip install -r requirements.txt
```

1.3 Quick Start

You can create your own beer-garden plugins without much problem at all. To start, we'll create the obligatory hello-world plugin. Creating a plugin is as simple as:

```
from brewtils.decorators import system, parameter, command
from brewtils.plugin import RemotePlugin

@system
class HelloWorld(object):

    @parameter(key="message", description="The message to echo", type="String")
    def say_hello(self, message="World!"):
        print("Hello, %s!" % message)
        return "Hello, %s!" % message

if __name__ == "__main__":
    client = HelloWorld()
    plugin = RemotePlugin(client,
                          name="hello",
                          version="0.0.1",
                          bg_host='127.0.0.1',
                          bg_port=2337)
    plugin.run()
```

Assuming you have a Beer Garden running on port 2337 on localhost, running this will register and start your plugin! You now have your first plugin running in beer-garden. Let's use another part of the `brewtils` library to exercise your plugin from python.

The `SystemClient` is designed to help you interact with registered Systems as if they were native Python objects.

```
from brewtils.rest.system_client import SystemClient

hello_client = SystemClient('localhost', 2337, 'hello')

request = hello_client.say_hello(message="from system client")

print(request.status) # 'SUCCESS'
print(request.output) # Hello, from system client!
```

In the background, the `SystemClient` has executed an HTTP POST with the payload required to get beer-garden to execute your command. The `SystemClient` is how most people interact with beer-garden when they are in the context of python and want to be making requests.

Of course, the rest of the API is accessible through the `brewtils` package. The `EasyClient` provides simple convenient methods to call the API and auto-serialize the responses. Suppose you want to get a list of all the commands on all systems:

```
from brewtils.rest.easy_client import EasyClient

client = EasyClient('localhost', 2337)

systems = client.find_systems()

for system in systems:
    for command in system.commands:
        print(command.name)
```

This is just a small taste of what is possible with the `EasyClient`. Feel free to explore all the methods that are exposed.

For more detailed information and better walkthroughs, checkout the full documentation!

1.4 Documentation

- Full Beer Garden documentation is available at https://beer-garden.io
- Brewtils Documentation is available at https://brewtils.readthedocs.io

CHAPTER 2

Installation

2.1 Stable release

To install Brewtils, run this command in your terminal:

```
$ pip install brewtils
```

This is the preferred method to install Brewtils, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for Brewtils can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git@github.com/beer-garden/brewtils.git
```

Or download the tarball:

```
$ curl  -OL https://github.com/beer-garden/brewtils/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

CHAPTER 4

Brewtils

Brewtils is the Python library for interfacing with Beergarden systems. If you are planning on writing beer-garden plugins, this is the correct library for you. In addition to writing plugins, it provides simple ways to query the API and is officially supported by the beer-garden team.

4.1 Features

Brewtils helps you interact with beer-garden.

- Easy way to create beer-garden plugins
- Full support of the entire Beer-Garden API
- Officially supported by the beer-garden team

4.2 Installation

To install brewtils, run this command in your terminal:

```
$ pip install brewtils
```

Or add it to your `requirements.txt`

```
$ cat brewtils >> requirements.txt
$ pip install -r requirements.txt
```

4.3 Quick Start

You can create your own beer-garden plugins without much problem at all. To start, we'll create the obligatory hello-world plugin. Creating a plugin is as simple as:

```
from brewtils.decorators import system, parameter, command
from brewtils.plugin import RemotePlugin

@system
class HelloWorld(object):

    @parameter(key="message", description="The message to echo", type="String")
    def say_hello(self, message="World!"):
        print("Hello, %s!" % message)
        return "Hello, %s!" % message

if __name__ == "__main__":
    client = HelloWorld()
    plugin = RemotePlugin(client,
                          name="hello",
                          version="0.0.1",
                          bg_host='127.0.0.1',
                          bg_port=2337)
    plugin.run()
```

Assuming you have a Beer Garden running on port 2337 on localhost, running this will register and start your plugin! You now have your first plugin running in beer-garden. Let's use another part of the `brewtils` library to exercise your plugin from python.

The `SystemClient` is designed to help you interact with registered Systems as if they were native Python objects.

```
from brewtils.rest.system_client import SystemClient

hello_client = SystemClient('localhost', 2337, 'hello')

request = hello_client.say_hello(message="from system client")

print(request.status) # 'SUCCESS'
print(request.output) # Hello, from system client!
```

In the background, the `SystemClient` has executed an HTTP POST with the payload required to get beer-garden to execute your command. The `SystemClient` is how most people interact with beer-garden when they are in the context of python and want to be making requests.

Of course, the rest of the API is accessible through the `brewtils` package. The `EasyClient` provides simple convenient methods to call the API and auto-serialize the responses. Suppose you want to get a list of all the commands on all systems:

```
from brewtils.rest.easy_client import EasyClient

client = EasyClient('localhost', 2337)

systems = client.find_systems()

for system in systems:
    for command in system.commands:
        print(command.name)
```

This is just a small taste of what is possible with the `EasyClient`. Feel free to explore all the methods that are exposed.

For more detailed information and better walkthroughs, checkout the full documentation!

4.4 Documentation

- Full Beer Garden documentation is available at https://beer-garden.io
- Brewtils Documentation is available at https://brewtils.readthedocs.io

CHAPTER 5

brewtils

5.1 brewtils package

5.1.1 Subpackages

brewtils.resolvers package

Submodules

brewtils.resolvers.gridfs module

brewtils.resolvers.parameter module

class brewtils.resolvers.parameter.**DownloadResolver**(*request*, *params_to_resolve*, *resolvers*, **_*)

Bases: *brewtils.resolvers.parameter.ParameterResolver*

resolve_parameters()

simple_resolve(*value*)

class brewtils.resolvers.parameter.**ParameterResolver**(*request*, *params_to_resolve*, *resolvers*)

Bases: object

Base class for parameter resolution.

This class is used under-the-hood for various plugin functions. Its purpose is to remove all the various cleanup and house keeping steps involved in resolving parameters. An example of an unresolved parameter is a dictionary which represents a bytes object. In this case, the user wants the open file descriptor, not the random dictionary that they don't know how to process. The parameter resolver helps handle these scenarios by providing the following API:

```
with ParameterResolver(request, ["my_file"], resolvers) as resolved_params:
    file_bytes = resolved_params["my_file"].read()
```

This is intended for internal use for the plugin class.

cleanup()

pre_resolve()

resolve_parameters()

simple_resolve(*value*)

class brewtils.resolvers.parameter.**UploadResolver**(*request*, *params_to_resolve*, *resolvers*)

Bases: *brewtils.resolvers.parameter.ParameterResolver*

simple_resolve(*value*)

Module contents

class brewtils.resolvers.**DownloadResolver**(*request*, *params_to_resolve*, *resolvers*, *_)

Bases: *brewtils.resolvers.parameter.ParameterResolver*

resolve_parameters()

simple_resolve(*value*)

class brewtils.resolvers.**FileResolver**(*client*)

Bases: object

Resolvers are meant to be written for specific storage types. In this case, we are uploading and downloading file chunks.

This class is meant to be used transparently to Plugin developers. Resolvers respond to two methods:

- *upload(value)*
- *download(bytes_parameter, writer)*

client
A *brewtils.EasyClient*

download(*file_id*, **args*)
Download the given bytes parameter.

Parameters **file_id** – A BG generated file ID

upload(*value*, ***kwargs*)
Upload the given value to the server if necessary.

The value can be one of the following:

1. A string representation of a valid filename.
2. An open file descriptor.

Parameters **value** – Value to upload.

Returns A valid beer garden assigned ID

class brewtils.resolvers.**UploadResolver**(*request*, *params_to_resolve*, *resolvers*)

Bases: *brewtils.resolvers.parameter.ParameterResolver*

simple_resolve(*value*)

brewtils.rest package

Submodules

brewtils.rest.client module

class brewtils.rest.client.**RestClient**(**args*, ***kwargs*)
Bases: object

HTTP client for communicating with Beer-garden.

The is the low-level client responsible for making the actual REST calls. Other clients (e.g. *brewtils.rest.easy_client.EasyClient*) build on this by providing useful abstractions.

Parameters

- **bg_host** (*str*) – Beer-garden hostname
- **bg_port** (*int*) – Beer-garden port
- **bg_url_prefix** (*str*) – URL path that will be used as a prefix when communicating with Beer-garden. Useful if Beer-garden is running on a URL other than '/'.
- **ssl_enabled** (*bool*) – Whether to use SSL for Beer-garden communication
- **ca_cert** (*str*) – Path to certificate file containing the certificate of the authority that issued the Beer-garden server certificate
- **ca_verify** (*bool*) – Whether to verify Beer-garden server certificate
- **client_cert** (*str*) – Path to client certificate to use when communicating with Beer-garden
- **api_version** (*int*) – Beer-garden API version to use
- **client_timeout** (*int*) – Max time to wait for Beer-garden server response
- **username** (*str*) – Username for Beer-garden authentication
- **password** (*str*) – Password for Beer-garden authentication
- **access_token** (*str*) – Access token for Beer-garden authentication
- **refresh_token** (*str*) – Refresh token for Beer-garden authentication

JSON_HEADERS = {'Accept': 'text/plain', 'Content-type': 'application/json'}

LATEST_VERSION = 1

can_connect(***kwargs*)
Determine if a connection to the Beer-garden server is possible

Parameters **kwargs** – Keyword arguments to pass to Requests session call

Returns A bool indicating if the connection attempt was successful. Will return False only if a ConnectionError is raised during the attempt. Any other exception will be re-raised.

Raises requests.exceptions.RequestException – The connection attempt resulted in an exception that indicates something other than a basic connection error. For example, an error with certificate verification.

delete_file(**args*, ***kwargs*)
Performs a GET on the specific File URL

Parameters

- **file_id** – File ID
- **kwargs** – Query parameters to be used in the GET request

Returns Requests Response object

delete_instance(**args*, ***kwargs*)
Performs a DELETE on an Instance URL

Parameters **instance_id** – Instance ID

Returns Requests Response object

delete_job(**args*, ***kwargs*)
Performs a DELETE on a Job URL

Parameters **job_id** – Job ID

Returns Requests Response object

delete_queue(**args*, ***kwargs*)
Performs a DELETE on a specific Queue URL

Parameters **queue_name** – Queue name

Returns Requests Response object

delete_queues(**args*, ***kwargs*)
Performs a DELETE on the Queues URL

Returns Requests Response object

delete_system(**args*, ***kwargs*)
Performs a DELETE on a System URL

Parameters **system_id** – System ID

Returns Requests Response object

get_command(**args*, ***kwargs*)
Performs a GET on the Command URL

Parameters **command_id** – Command ID

Returns Requests Response object

get_commands(**args*, ***kwargs*)
Performs a GET on the Commands URL

Returns Requests Response object

get_config(**args*, ***kwargs*)
Perform a GET to the config URL

Returns Requests Response object

get_file(**args*, ***kwargs*)
Performs a GET on the specific File URL

Parameters

- **file_id** – File ID

- **kwargs** – Query parameters to be used in the GET request

Returns Requests Response object

get_instance(**args*, ***kwargs*)
Performs a GET on the Instance URL

Parameters **instance_id** – Instance ID

Returns Requests Response object

get_job(**args*, ***kwargs*)
Performs a GET on the Job URL

Parameters **job_id** – Job ID

Returns Requests Response object

get_jobs(**args*, ***kwargs*)
Performs a GET on the Jobs URL.

Parameters **kwargs** – Query parameters to be used in the GET request

Returns Requests Response object

get_logging_config(**args*, ***kwargs*)
Perform a GET to the logging config URL

Parameters **kwargs** – Query parameters to be used in the GET request

Returns Requests Response object

get_queues(**args*, ***kwargs*)
Performs a GET on the Queues URL

Returns Requests Response object

get_request(**args*, ***kwargs*)
Performs a GET on the Request URL

Parameters **request_id** – Request ID

Returns Requests Response object

get_requests(**args*, ***kwargs*)
Performs a GET on the Requests URL

Parameters **kwargs** – Query parameters to be used in the GET request

Returns Requests Response object

get_system(**args*, ***kwargs*)
Performs a GET on the System URL

Parameters

- **system_id** – System ID
- **kwargs** – Query parameters to be used in the GET request

Returns Requests Response object

get_systems(**args*, ***kwargs*)
Perform a GET on the System collection URL

Parameters **kwargs** – Query parameters to be used in the GET request

Returns Requests Response object

get_tokens(*username=None*, *password=None*)
Use a username and password to get access and refresh tokens

Parameters

- **username** – Beergarden username
- **password** – Beergarden password

Returns Requests Response object

get_user(**args*, ***kwargs*)
Performs a GET on the specific User URL

Parameters **user_identifier** – User ID or username

Returns Requests Response object

get_version(**args*, ***kwargs*)
Perform a GET to the version URL

Returns Requests Response object

patch_instance(**args*, ***kwargs*)
Performs a PATCH on the instance URL

Parameters

- **instance_id** – Instance ID
- **payload** – Serialized PatchOperation

Returns Requests Response object

patch_job(**args*, ***kwargs*)
Performs a PATCH on the Job URL

Parameters

- **job_id** – Job ID
- **payload** – Serialized PatchOperation

Returns Requests Response object

patch_request(**args*, ***kwargs*)
Performs a PATCH on the Request URL

Parameters

- **request_id** – Request ID
- **payload** – Serialized PatchOperation

Returns Requests Response object

patch_system(**args*, ***kwargs*)
Performs a PATCH on a System URL

Parameters

- **system_id** – System ID
- **payload** – Serialized PatchOperation

Returns Requests Response object

post_event(**args*, ***kwargs*)
Performs a POST on the event URL

Parameters

- **payload** – Serialized new event definition
- **publishers** – Array of publishers to use

Returns Requests Response object

post_file(**args*, ***kwargs*)
Performs a POST on the file URL.

Parameters

- **fd** – A file descriptor
- **file_params** – Metadata about the file
- **current_position** – The current cursor position for the file object

Returns A Requests Response object

post_jobs(**args*, ***kwargs*)
Performs a POST on the Job URL

Parameters **payload** – New Job definition

Returns Requests Response object

post_requests(**args*, ***kwargs*)
Performs a POST on the Request URL

Parameters

- **payload** – New Request definition
- **kwargs** – Extra request parameters

Keyword Arguments

- **blocking** – Wait for request to complete
- **timeout** – Maximum seconds to wait

Returns Requests Response object

post_systems(**args*, ***kwargs*)
Performs a POST on the System URL

Parameters **payload** – New System definition

Returns Requests Response object

refresh(*refresh_token=None*)
Use a refresh token to obtain a new access token

Parameters **refresh_token** – Refresh token to use

Returns Requests Response object

class brewtils.rest.client.**TimeoutAdapter**(***kwargs*)
Bases: requests.adapters.HTTPAdapter

Transport adapter with a default request timeout

send(**args*, ***kwargs*)
Sends PreparedRequest object. Returns Response object.

Parameters

- **request** – The PreparedRequest being sent.
- **stream** – (optional) Whether to stream the request content.
- **timeout** (*float or tuple or urllib3 Timeout object*) – (optional) How long to wait for the server to send data before giving up, as a float, or a (connect timeout, read timeout) tuple.
- **verify** – (optional) Either a boolean, in which case it controls whether we verify the server's TLS certificate, or a string, in which case it must be a path to a CA bundle to use
- **cert** – (optional) Any user-provided SSL certificate to be trusted.
- **proxies** – (optional) The proxies dictionary to apply to the request.

Return type requests.Response

brewtils.rest.client.**enable_auth**(*method*)
Decorate methods with this to enable using authentication

brewtils.rest.easy_client module

class brewtils.rest.easy_client.**EasyClient**(***kwargs*)
Bases: object

Client for simplified communication with Beergarden

This class is intended to be a middle ground between the RestClient and SystemClient. It provides a 'cleaner' interface to some common Beergarden operations than is exposed by the lower-level RestClient. On the other hand, the SystemClient is much better for generating Beergarden Requests.

Parameters

- **bg_host** (*str*) – Beer-garden hostname
- **bg_port** (*int*) – Beer-garden port
- **bg_url_prefix** (*str*) – URL path that will be used as a prefix when communicating with Beer-garden. Useful if Beer-garden is running on a URL other than '/'.
- **ssl_enabled** (*bool*) – Whether to use SSL for Beer-garden communication
- **ca_cert** (*str*) – Path to certificate file containing the certificate of the authority that issued the Beer-garden server certificate
- **ca_verify** (*bool*) – Whether to verify Beer-garden server certificate
- **client_cert** (*str*) – Path to client certificate to use when communicating with Beer-garden
- **api_version** (*int*) – Beer-garden API version to use
- **client_timeout** (*int*) – Max time to wait for Beer-garden server response
- **username** (*str*) – Username for Beer-garden authentication
- **password** (*str*) – Password for Beer-garden authentication
- **access_token** (*str*) – Access token for Beer-garden authentication
- **refresh_token** (*str*) – Refresh token for Beer-garden authentication

can_connect(***kwargs*)
Determine if the Beergarden server is responding.

Kwargs: Arguments passed to the underlying Requests method

Returns A bool indicating if the connection attempt was successful. Will return False only if a ConnectionError is raised during the attempt. Any other exception will be re-raised.

Raises `requests.exceptions.RequestException` – The connection attempt resulted in an exception that indicates something other than a basic connection error. For example, an error with certificate verification.

clear_all_queues()
Cancel and remove all Requests in all queues

Returns True if the clear was successful

Return type bool

clear_queue(*queue_name*)
Cancel and remove all Requests from a message queue

Parameters **queue_name** (*str*) – The name of the queue to clear

Returns True if the clear was successful

Return type bool

create_job(*job*)
Create a new Job

Parameters **job** (Job) – New Job definition

Returns The newly-created Job

Return type *Job*

create_request(*request*, ***kwargs*)
Create a new Request

Parameters

- **request** – New request definition
- **kwargs** – Extra request parameters

Keyword Arguments

- **blocking** (*bool*) – Wait for request to complete before returning
- **timeout** (*int*) – Maximum seconds to wait for completion

Returns The newly-created Request

Return type *Request*

create_system(*system*)
Create a new System

Parameters **system** (System) – The System to create

Returns The newly-created system

Return type *System*

delete_file(*file_id*)
Delete a given file on the Beer Garden server.

Parameters **file_id** – The beer garden-assigned file id.

Returns The API response

download_file(*file_id*)
Download a file from the Beer Garden server.

Parameters **file_id** – The beer garden-assigned file id.

Returns A file object

find_jobs(**kwargs*)
Find Jobs using keyword arguments as search parameters

Parameters ****kwargs** – Search parameters

Returns List of Jobs matching the search parameters

Return type List[*Job*]

find_requests(**kwargs*)
Find Requests using keyword arguments as search parameters

Parameters ****kwargs** – Search parameters

Returns List of Systems matching the search parameters

Return type List[*Request*]

find_systems(**kwargs*)
Find Systems using keyword arguments as search parameters

Parameters ****kwargs** – Search parameters

Returns List of Systems matching the search parameters

Return type List[*System*]

find_unique_request(**kwargs*)
Find a unique request

Note: If 'id' is a given keyword argument then all other parameters will be ignored.

Parameters ****kwargs** – Search parameters

Returns The Request if found, None otherwise

Return type *Request*, None

Raises FetchError – More than one matching Request was found

find_unique_system(**kwargs*)
Find a unique system

Note: If 'id' is a given keyword argument then all other parameters will be ignored.

Parameters ****kwargs** – Search parameters

Returns The System if found, None otherwise

Return type *System*, None

Raises FetchError – More than one matching System was found

get_config()
Get configuration

Returns Configuration dictionary

Return type dict

get_instance(*instance_id*)
Get an Instance

Parameters **instance_id** – The Id

Returns The Instance

get_instance_status(*instance_id*)
Get an Instance's status

Parameters **instance_id** – The Id

Returns The Instance's status

get_logging_config(*system_name=None*, *local=False*)
Get a logging configuration

Note that the system_name is not relevant and is only provided for backward-compatibility.

Parameters **system_name** (*str*) – UNUSED

Returns The configuration object

Return type dict

get_queues()
Retrieve all queue information

Returns List of all Queues

Return type List[*Queue*]

get_request(*request_id*)
Get a Request

Parameters **request_id** – The Id

Returns The Request

get_system(*system_id*, ***kwargs*)
Get a System

Parameters **system_id** – The Id

Returns The System

get_user(*user_identifier*)
Find a user

Parameters **user_identifier** (*str*) – User ID or username

Returns The User

Return type *Principal*

get_version(***kwargs*)
Get Bartender, Brew-view, and API version information

Parameters ****kwargs** – Extra parameters

Returns Response object with version information in the body

Return type dict

initialize_instance(*instance_id*, *runner_id=None*)
Start an Instance

Parameters

- **instance_id** (*str*) – The Instance ID
- **runner_id** (*str*) – The PluginRunner ID, if any

Returns The updated Instance

Return type *Instance*

instance_heartbeat(*instance_id*)
Send an Instance heartbeat

Parameters **instance_id** (*str*) – The Instance ID

Returns True if the heartbeat was successful

Return type bool

pause_job(*job_id*)
Pause a Job

Parameters **job_id** (*str*) – The Job ID

Returns The updated Job

Return type *Job*

publish_event(**args*, ***kwargs*)
Publish a new event

Parameters

- ***args** – If a positional argument is given it's assumed to be an Event and will be used
- ****kwargs** – Will be used to construct a new Event to publish if no Event is given in the positional arguments

Keyword Arguments **_publishers** (*Optional[List[str]]*) – List of publisher names. If given the Event will only be published to the specified publishers. Otherwise all publishers known to Beergarden will be used.

Returns True if the publish was successful

Return type bool

remove_instance(*instance_id*)
Remove an Instance

Parameters **instance_id** (*str*) – The Instance ID

Returns True if the remove was successful

Return type bool

remove_job(*job_id*)
Remove a unique Job

Parameters **job_id** (*str*) – The Job ID

Returns True if removal was successful

Return type bool

Raises `DeleteError` – Couldn't remove Job

remove_system(***kwargs*)
Remove a unique System

Parameters ****kwargs** – Search parameters

Returns True if removal was successful

Return type bool

Raises `FetchError` – Couldn't find a System matching given parameters

resume_job(*job_id*)
Resume a Job

Parameters **job_id** (*str*) – The Job ID

Returns The updated Job

Return type *Job*

update_instance(*instance_id*, ***kwargs*)
Update an Instance status

Parameters **instance_id** (*str*) – The Instance ID

Keyword Arguments

- **new_status** (*str*) – The new status
- **metadata** (*dict*) – Will be added to existing instance metadata

Returns The updated Instance

Return type *Instance*

update_instance_status(*instance_id*, *new_status*)
Update an Instance status

Parameters

- **instance_id** (*str*) – The Instance ID
- **new_status** (*str*) – The new status

Returns The updated Instance

Return type *Instance*

update_request(*request_id*, *status=None*, *output=None*, *error_class=None*)
Update a Request

Parameters

- **request_id** (*str*) – The Request ID
- **status** (*Optional[str]*) – New Request status
- **output** (*Optional[str]*) – New Request output
- **error_class** (*Optional[str]*) – New Request error class

Returns The updated response

Return type Response

update_system(*system_id*, *new_commands=None*, ***kwargs*)
Update a System

Parameters

- **system_id** (*str*) – The System ID
- **new_commands** (*Optional[List[Command]]*) – New System commands

Keyword Arguments

- **add_instance** (Instance) – An Instance to append
- **metadata** (*dict*) – New System metadata
- **description** (*str*) – New System description
- **display_name** (*str*) – New System display name
- **icon_name** (*str*) – New System icon name

Returns The updated system

Return type *System*

upload_file(*file_to_upload*, *desired_filename=None*, *file_params=None*)
Upload a given file to the Beer Garden server.

Parameters

- **file_to_upload** – Can either be an open file descriptor or a path.
- **desired_filename** – The desired filename, if none is provided it
- **use the basename of the file_to_upload** (*will*) –
- **file_params** – The metadata surrounding the file. Valid Keys: See brewtils File model

Returns A BG file ID.

who_am_i()
Find user using the current set of credentials

Returns The User

Return type *Principal*

brewtils.rest.easy_client.**get_easy_client**(***kwargs*)
Easy way to get an EasyClient

The benefit to this method over creating an EasyClient directly is that this method will also search the environment for parameters. Kwargs passed to this method will take priority, however.

Parameters ****kwargs** – Options for configuring the EasyClient

Returns The configured client

Return type *brewtils.rest.easy_client.EasyClient*

brewtils.rest.easy_client.**handle_response_failure**(*response*, *default_exc=<class 'brewtils.errors.RestError'>*, *raise_404=True*)
Deal with a response with non-2xx status code

Parameters

- **response** – The response object
- **default_exc** – The exception to raise if no specific exception is warranted
- **raise_404** – If True a response with status code 404 will raise a NotFoundError. If False the method will return None.

Returns None - this function will always raise

Raises

- `NotFoundError` – Status code 404 and raise_404 is True
- `WaitExceededError` – Status code 408
- `ConflictError` – Status code 409
- `TooLargeError` – Status code 413
- `ValidationError` – Any other 4xx status codes
- `RestConnectionError` – Status code 503
- `default_exc` – Any other status code

`brewtils.rest.easy_client.`**`wrap_response`**(*return_boolean=False*, *parse_method=None*, *parse_many=False*, *default_exc=<class 'brewtils.errors.RestError'>*, *raise_404=True*)

Decorator to consolidate response parsing and error handling

Parameters

- **`return_boolean`** – If True, a successful response will also return True
- **`parse_method`** – Response json will be passed to this method of the SchemaParser
- **`parse_many`** – Will be passed as the 'many' parameter when parsing the response
- **`default_exc`** – Will be passed to handle_response_failure for failed responses
- **`raise_404`** – Will be passed to handle_response_failure for failed responses

Returns

- True if return_boolean is True and the response status code is 2xx.
- The response object if return_boolean is False and parse_method is ""
- A parsed Brewtils model if return_boolean is False and parse_method is defined

Raises `RestError` – The response has a non-2xx status code. Note that the specific exception raised depends on the response status code and the argument passed as the default_exc parameter.

brewtils.rest.system_client module

`class` `brewtils.rest.system_client.`**`SystemClient`**(***kwargs*)

Bases: `object`

High-level client for generating requests for a Beer-garden System.

SystemClient creation: This class is intended to be the main way to create Beer-garden requests. Create an instance with Beer-garden connection information and a system name:

```
client = SystemClient(
    system_name='example_system',
    system_namespace='default',
    bg_host="host",
    bg_port=2337,
)
```

Note: Passing an empty string as the system_namespace parameter will evalutate to the local garden's default namespace.

Pass additional keyword arguments for more granularity:

version_constraint: Allows specifying a particular system version. Can be a version literal ('1.0.0') or the special value 'latest.' Using 'latest' will allow the SystemClient to retry a request if it fails due to a missing system (see Creating Requests).

default_instance: The instance name to use when creating a request if no other instance name is specified. Since each request must be addressed to a specific instance this is a convenience to prevent needing to specify the instance for each request.

always_update: If True the SystemClient will always attempt to reload the system definition before making a request. This is useful to ensure Requests are always made against the latest version of the system. If not set the System definition will be loaded when making the first request and will only be reloaded if a Request fails.

Loading the System: The System definition is lazily loaded, so nothing happens until the first attempt to send a Request. At that point the SystemClient will query Beer-garden to get a system definition that matches the system_name and version_constraint. If no matching System can be found a FetchError will be raised. If always_update was set to True this will happen before making each request, not only the first.

Making a Request: The standard way to create and send requests is by calling object attributes:

```
request = client.example_command(param_1='example_param')
```

In the normal case this will block until the request completes. Request completion is determined by periodically polling Beer-garden to check the Request status. The time between polling requests starts at 0.5s and doubles each time the request has still not completed, up to max_delay. If a timeout was specified and the Request has not completed within that time a `ConnectionTimeoutError` will be raised.

It is also possible to create the SystemClient in non-blocking mode by specifying blocking=False. In this case the request creation will immediately return a Future and will spawn a separate thread to poll for Request completion. The max_concurrent parameter is used to control the maximum threads available for polling.

```
# Create a SystemClient with blocking=False
client = SystemClient(
    system_name='example_system',
    system_namespace='default',
    bg_host="localhost",
    bg_port=2337,
    blocking=False,
)

# Create and send 5 requests without waiting for request completion
futures = [client.example_command(param_1=number) for number in range(5)]

# Now wait on all requests to complete
concurrent.futures.wait(futures)
```

If the request creation process fails (e.g. the command failed validation) and version_constraint is 'latest' then the SystemClient will check to see if a newer version is available, and if so it will attempt to make the request on that version. This is so users of the SystemClient that don't necessarily care about the target system version don't need to be restarted every time the target system is updated.

It's also possible to control what happens when a Request results in an ERROR. If the `raise_on_error` parameter is set to False (the default) then Requests that are not successful simply result in a Request with a status of ERROR, and it is the plugin developer's responsibility to check for this case. However, if

`raise_on_error` is set to True then this will result in a `RequestFailedError` being raised. This will happen regardless of the value of the `blocking` flag.

Tweaking Beer-garden Request Parameters: There are several parameters that control how beer-garden routes / processes a request. To denote these as intended for Beer-garden itself (rather than a parameter to be passed to the Plugin) prepend a leading underscore to the argument name.

Sending to another instance:

```
request = client.example_command(
    _instance_name="instance_2", param_1="example_param"
)
```

Request with a comment:

```
request = client.example_command(
    _comment="I'm a beer-garden comment!", param_1="example_param"
)
```

Without the leading underscore the arguments would be treated the same as "param_1" - another parameter to be passed to the plugin.

Request that raises:

```
client = SystemClient(
    system_name="foo",
    system_namespace='default',
    bg_host="localhost",
    bg_port=2337,
)

try:
    client.command_that_errors(_raise_on_error=True)
except RequestFailedError:
    print("I could have just ignored this")
```

Parameters

- **system_name** (*str*) – Name of the System to make Requests on
- **system_namespace** (*str*) – Namespace of the System to make Requests on
- **version_constraint** (*str*) – System version to make Requests on. Can be specific ('1.0.0') or 'latest'.
- **default_instance** (*str*) – Name of the Instance to make Requests on
- **always_update** (*bool*) – Whether to check if a newer version of the System exists before making each Request. Only relevant if `version_constraint='latest'`
- **timeout** (*int*) – Seconds to wait for a request to complete. 'None' means wait forever.
- **max_delay** (*int*) – Maximum number of seconds to wait between status checks for a created request
- **blocking** (*bool*) – Flag indicating whether creation will block until the Request is complete or return a Future that will complete when the Request does
- **max_concurrent** (*int*) – Maximum number of concurrent requests allowed. Only has an effect when blocking=False.

- **raise_on_error** (*bool*) – Flag controlling whether created Requests that complete with an ERROR state should raise an exception
- **bg_host** (*str*) – Beer-garden hostname
- **bg_port** (*int*) – Beer-garden port
- **bg_url_prefix** (*str*) – URL path that will be used as a prefix when communicating with Beer-garden. Useful if Beer-garden is running on a URL other than '/'.
- **ssl_enabled** (*bool*) – Whether to use SSL for Beer-garden communication
- **ca_cert** (*str*) – Path to certificate file containing the certificate of the authority that issued the Beer-garden server certificate
- **ca_verify** (*bool*) – Whether to verify Beer-garden server certificate
- **client_cert** (*str*) – Path to client certificate to use when communicating with Beer-garden
- **api_version** (*int*) – Beer-garden API version to use
- **client_timeout** (*int*) – Max time to wait for Beer-garden server response
- **username** (*str*) – Username for Beer-garden authentication
- **password** (*str*) – Password for Beer-garden authentication
- **access_token** (*str*) – Access token for Beer-garden authentication
- **refresh_token** (*str*) – Refresh token for Beer-garden authentication

create_bg_request(*command_name*, ***kwargs*)

Create a callable that will execute a Beer-garden request when called.

Normally you interact with the SystemClient by accessing attributes, but there could be certain cases where you want to create a request without sending it.

Example:

```
client = SystemClient(host, port, 'system', blocking=False)

# Create two callables - one with a parameter and one without
uncreated_requests = [
    client.create_bg_request('command_1', arg_1='Hi!'),
    client.create_bg_request('command_2'),
]

# Calling creates and sends the request
# The result of each is a future because blocking=False on the SystemClient
futures = [req() for req in uncreated_requests]

# Wait for all the futures to complete
concurrent.futures.wait(futures)
```

Parameters

- **command_name** (*str*) – Name of the Command to send
- **kwargs** (*dict*) – Will be passed as parameters when creating the Request

Returns Partial that will create and execute a Beer-garden request when called

Raises AttributeError – System does not have a Command with the given command_name

load_bg_system()
Query beer-garden for a System definition

This method will make the query to beer-garden for a System matching the name and version constraints specified during SystemClient instance creation.

If this method completes successfully the SystemClient will be ready to create and send Requests.

Returns None

Raises FetchError – Unable to find a matching System

send_bg_request(**args*, ***kwargs*)
Actually create a Request and send it to Beer-garden

Note: This method is intended for advanced use only, mainly cases where you're using the SystemClient without a predefined System. It assumes that everything needed to construct the request is being passed in kwargs. If this doesn't sound like what you want you should check out create_bg_request.

Parameters

- **args** (*list*) – Unused. Passing positional parameters indicates a bug
- **kwargs** (*dict*) – All necessary request parameters, including Beer-garden internal parameters

Returns A completed Request object blocking=False: A future that will be completed when the Request does

Return type blocking=True

Raises ValidationError – Request creation failed validation on the server

Module contents

brewtils.rest.**normalize_url_prefix**(*url_prefix*)
Enforce a consistent URL representation

The normalized prefix will begin and end with '/'. If there is no prefix the normalized form will be '/'.

Examples

INPUT NORMALIZED None '/' '' '/' '/' '/' 'example' '/example/' '/example' '/example/' 'example/' '/example/' '/example/' '/example/'

Parameters **url_prefix** (*str*) – The prefix

Returns The normalized prefix

Return type str

brewtils.test package

Submodules

brewtils.test.comparable module

Module to simplify model comparisons.

WARNING: This module was created to simplify testing. As such, it's not recommended for production use.

ANOTHER WARNING: This module subject to change outside of the normal deprecation cycle.

Seriously, this is a 'use at your own risk' kind of thing.

brewtils.test.comparable.**assert_instance_equal**()

Wrapper that will translate AssertionError to a boolean.

This is a safety measure in case these functions are used outside of a testing context. This isn't recommended, but naked asserts are still unacceptable in any packaged code. This method will translate the various comparison functions to a simple boolean return.

Note that in a testing context the AssertionError is re-raised. This is because it's much more helpful to know the specific assertion that failed, as it could be something nested several levels deep.

Parameters

- **obj1** – Passed through to _assert_equal
- **obj2** – Passed through to _assert_equal
- **do_raise** – If True, re-raise any raised AssertionError. This helps with nested comparisons.
- ****kwargs** – Passed through to _assert_equal

Returns

True if the comparison was equal. False if:

- The comparison was not equal and
- do_raise is False and
- called from outside of a testing context

Raises `AssertionError` – The comparison was not equal. Assertion will be translated to a boolean False if do_raise is False and called from outside of a testing context.

brewtils.test.comparable.**assert_choices_equal**()

Wrapper that will translate AssertionError to a boolean.

This is a safety measure in case these functions are used outside of a testing context. This isn't recommended, but naked asserts are still unacceptable in any packaged code. This method will translate the various comparison functions to a simple boolean return.

Note that in a testing context the AssertionError is re-raised. This is because it's much more helpful to know the specific assertion that failed, as it could be something nested several levels deep.

Parameters

- **obj1** – Passed through to _assert_equal
- **obj2** – Passed through to _assert_equal
- **do_raise** – If True, re-raise any raised AssertionError. This helps with nested comparisons.
- ****kwargs** – Passed through to _assert_equal

Returns

True if the comparison was equal. False if:

- The comparison was not equal and
- do_raise is False and
- called from outside of a testing context

Raises `AssertionError` – The comparison was not equal. Assertion will be translated to a boolean False if do_raise is False and called from outside of a testing context.

`brewtils.test.comparable.`**`assert_patch_equal`**`()`

Wrapper that will translate AssertionError to a boolean.

This is a safety measure in case these functions are used outside of a testing context. This isn't recommended, but naked asserts are still unacceptable in any packaged code. This method will translate the various comparison functions to a simple boolean return.

Note that in a testing context the AssertionError is re-raised. This is because it's much more helpful to know the specific assertion that failed, as it could be something nested several levels deep.

Parameters

- **`obj1`** – Passed through to _assert_equal
- **`obj2`** – Passed through to _assert_equal
- **`do_raise`** – If True, re-raise any raised AssertionError. This helps with nested comparisons.
- **`**kwargs`** – Passed through to _assert_equal

Returns

True if the comparison was equal. False if:

- The comparison was not equal and
- do_raise is False and
- called from outside of a testing context

Raises `AssertionError` – The comparison was not equal. Assertion will be translated to a boolean False if do_raise is False and called from outside of a testing context.

`brewtils.test.comparable.`**`assert_logging_config_equal`**`()`

Wrapper that will translate AssertionError to a boolean.

This is a safety measure in case these functions are used outside of a testing context. This isn't recommended, but naked asserts are still unacceptable in any packaged code. This method will translate the various comparison functions to a simple boolean return.

Note that in a testing context the AssertionError is re-raised. This is because it's much more helpful to know the specific assertion that failed, as it could be something nested several levels deep.

Parameters

- **`obj1`** – Passed through to _assert_equal
- **`obj2`** – Passed through to _assert_equal
- **`do_raise`** – If True, re-raise any raised AssertionError. This helps with nested comparisons.
- **`**kwargs`** – Passed through to _assert_equal

Returns

True if the comparison was equal. False if:

- The comparison was not equal and
- do_raise is False and
- called from outside of a testing context

Raises `AssertionError` – The comparison was not equal. Assertion will be translated to a boolean False if do_raise is False and called from outside of a testing context.

`brewtils.test.comparable.`**`assert_event_equal`**(*obj1*, *obj2*, *do_raise=False*)

`brewtils.test.comparable.`**`assert_queue_equal`**()

Wrapper that will translate AssertionError to a boolean.

This is a safety measure in case these functions are used outside of a testing context. This isn't recommended, but naked asserts are still unacceptable in any packaged code. This method will translate the various comparison functions to a simple boolean return.

Note that in a testing context the AssertionError is re-raised. This is because it's much more helpful to know the specific assertion that failed, as it could be something nested several levels deep.

Parameters

- **`obj1`** – Passed through to _assert_equal
- **`obj2`** – Passed through to _assert_equal
- **`do_raise`** – If True, re-raise any raised AssertionError. This helps with nested comparisons.
- **`**kwargs`** – Passed through to _assert_equal

Returns

True if the comparison was equal. False if:

- The comparison was not equal and
- do_raise is False and
- called from outside of a testing context

Raises `AssertionError` – The comparison was not equal. Assertion will be translated to a boolean False if do_raise is False and called from outside of a testing context.

`brewtils.test.comparable.`**`assert_request_template_equal`**()

Wrapper that will translate AssertionError to a boolean.

This is a safety measure in case these functions are used outside of a testing context. This isn't recommended, but naked asserts are still unacceptable in any packaged code. This method will translate the various comparison functions to a simple boolean return.

Note that in a testing context the AssertionError is re-raised. This is because it's much more helpful to know the specific assertion that failed, as it could be something nested several levels deep.

Parameters

- **`obj1`** – Passed through to _assert_equal
- **`obj2`** – Passed through to _assert_equal
- **`do_raise`** – If True, re-raise any raised AssertionError. This helps with nested comparisons.

- ****kwargs** – Passed through to _assert_equal

Returns

True if the comparison was equal. False if:

- The comparison was not equal and
- do_raise is False and
- called from outside of a testing context

Raises `AssertionError` – The comparison was not equal. Assertion will be translated to a boolean False if do_raise is False and called from outside of a testing context.

`brewtils.test.comparable.`**`assert_trigger_equal`**`()`

Wrapper that will translate AssertionError to a boolean.

This is a safety measure in case these functions are used outside of a testing context. This isn't recommended, but naked asserts are still unacceptable in any packaged code. This method will translate the various comparison functions to a simple boolean return.

Note that in a testing context the AssertionError is re-raised. This is because it's much more helpful to know the specific assertion that failed, as it could be something nested several levels deep.

Parameters

- **obj1** – Passed through to _assert_equal
- **obj2** – Passed through to _assert_equal
- **do_raise** – If True, re-raise any raised AssertionError. This helps with nested comparisons.
- ****kwargs** – Passed through to _assert_equal

Returns

True if the comparison was equal. False if:

- The comparison was not equal and
- do_raise is False and
- called from outside of a testing context

Raises `AssertionError` – The comparison was not equal. Assertion will be translated to a boolean False if do_raise is False and called from outside of a testing context.

`brewtils.test.comparable.`**`assert_command_equal`**`(`*obj1*, *obj2*, *do_raise=False*`)`

`brewtils.test.comparable.`**`assert_parameter_equal`**`(`*obj1*, *obj2*, *do_raise=False*`)`

`brewtils.test.comparable.`**`assert_principal_equal`**`(`*obj1*, *obj2*, *do_raise=False*`)`

`brewtils.test.comparable.`**`assert_request_equal`**`(`*obj1*, *obj2*, *do_raise=False*`)`

Assert that two requests are 'equal'.

This is the most complicated due to how we serialize parent and children requests to avoid reference loops.

Parent fields will not serialize their children. That's why compare_parent asserts that the children field is None.

The requests in the children field will not serialize their parents or children. That's why compare_child asserts that both the parent and children fields are None.

`brewtils.test.comparable.`**`assert_role_equal`**`(`*obj1*, *obj2*, *do_raise=False*`)`

`brewtils.test.comparable.`**`assert_system_equal`**`(`*obj1*, *obj2*, *do_raise=False*`)`

brewtils.test.comparable.**assert_job_equal**(*obj1*, *obj2*, *do_raise=False*)

brewtils.test.comparable.**assert_request_file_equal**()
Wrapper that will translate AssertionError to a boolean.

This is a safety measure in case these functions are used outside of a testing context. This isn't recommended, but naked asserts are still unacceptable in any packaged code. This method will translate the various comparison functions to a simple boolean return.

Note that in a testing context the AssertionError is re-raised. This is because it's much more helpful to know the specific assertion that failed, as it could be something nested several levels deep.

Parameters

- **obj1** – Passed through to _assert_equal
- **obj2** – Passed through to _assert_equal
- **do_raise** – If True, re-raise any raised AssertionError. This helps with nested comparisons.
- ****kwargs** – Passed through to _assert_equal

Returns

True if the comparison was equal. False if:

- The comparison was not equal and
- do_raise is False and
- called from outside of a testing context

Raises AssertionError – The comparison was not equal. Assertion will be translated to a boolean False if do_raise is False and called from outside of a testing context.

brewtils.test.comparable.**assert_operation_equal**(*obj1*, *obj2*, *do_raise=False*)

brewtils.test.comparable.**assert_runner_equal**()
Wrapper that will translate AssertionError to a boolean.

This is a safety measure in case these functions are used outside of a testing context. This isn't recommended, but naked asserts are still unacceptable in any packaged code. This method will translate the various comparison functions to a simple boolean return.

Note that in a testing context the AssertionError is re-raised. This is because it's much more helpful to know the specific assertion that failed, as it could be something nested several levels deep.

Parameters

- **obj1** – Passed through to _assert_equal
- **obj2** – Passed through to _assert_equal
- **do_raise** – If True, re-raise any raised AssertionError. This helps with nested comparisons.
- ****kwargs** – Passed through to _assert_equal

Returns

True if the comparison was equal. False if:

- The comparison was not equal and
- do_raise is False and
- called from outside of a testing context

Raises `AssertionError` – The comparison was not equal. Assertion will be translated to a boolean False if do_raise is False and called from outside of a testing context.

brewtils.test.fixtures module

`brewtils.test.fixtures.`**`bg_choices`**(**args*, ***kwargs*)

`brewtils.test.fixtures.`**`bg_command`**(**args*, ***kwargs*)
Use the bg_command fixture instead.

`brewtils.test.fixtures.`**`bg_command_2`**(**args*, ***kwargs*)
Use the bg_command fixture instead.

`brewtils.test.fixtures.`**`bg_cron_job`**(**args*, ***kwargs*)
A beer garden cron job

`brewtils.test.fixtures.`**`bg_cron_trigger`**(**args*, ***kwargs*)
A cron trigger as a model.

`brewtils.test.fixtures.`**`bg_date_trigger`**(**args*, ***kwargs*)
A date trigger as a model.

`brewtils.test.fixtures.`**`bg_event`**(**args*, ***kwargs*)
An event as a model.

`brewtils.test.fixtures.`**`bg_garden`**(**args*, ***kwargs*)
An operation as a model.

`brewtils.test.fixtures.`**`bg_instance`**(**args*, ***kwargs*)
An instance as a model.

`brewtils.test.fixtures.`**`bg_interval_job`**(**args*, ***kwargs*)
A beer garden interval job

`brewtils.test.fixtures.`**`bg_interval_trigger`**(**args*, ***kwargs*)
An interval trigger as a model.

`brewtils.test.fixtures.`**`bg_job`**(**args*, ***kwargs*)
A job as a model.

`brewtils.test.fixtures.`**`bg_logging_config`**(**args*, ***kwargs*)
A logging config as a model.

`brewtils.test.fixtures.`**`bg_operation`**(**args*, ***kwargs*)
An operation as a model.

`brewtils.test.fixtures.`**`bg_parameter`**(**args*, ***kwargs*)
Parameter based on the parameter_dict

`brewtils.test.fixtures.`**`bg_patch`**(**args*, ***kwargs*)
A patch as a model.

`brewtils.test.fixtures.`**`bg_patch2`**(**args*, ***kwargs*)
A patch as a model.

`brewtils.test.fixtures.`**`bg_principal`**(**args*, ***kwargs*)

`brewtils.test.fixtures.`**`bg_queue`**(**args*, ***kwargs*)
A queue as a model.

`brewtils.test.fixtures.`**`bg_request`**(**args*, ***kwargs*)
A request as a model.

brewtils.test.fixtures.**bg_request_file**(**args*, ***kwargs*)
A request file as a model

brewtils.test.fixtures.**bg_request_template**(**args*, ***kwargs*)
Request template as a bg model.

brewtils.test.fixtures.**bg_role**(**args*, ***kwargs*)

brewtils.test.fixtures.**bg_runner**(**args*, ***kwargs*)
A runner as a model.

brewtils.test.fixtures.**bg_system**(**args*, ***kwargs*)
A system as a model.

brewtils.test.fixtures.**child_request**(**args*, ***kwargs*)
A child request as a model.

brewtils.test.fixtures.**child_request_dict**(**args*, ***kwargs*)
A child request represented as a dictionary.

brewtils.test.fixtures.**choices_dict**(**args*, ***kwargs*)
Choices as a dictionary.

brewtils.test.fixtures.**command_dict**(**args*, ***kwargs*)
A command represented as a dictionary.

brewtils.test.fixtures.**command_dict_2**(**args*, ***kwargs*)
A second command represented as a dictionary.

brewtils.test.fixtures.**cron_job_dict**(**args*, ***kwargs*)
A cron job represented as a dictionary.

brewtils.test.fixtures.**cron_trigger_dict**(**args*, ***kwargs*)
A cron trigger as a dictionary.

brewtils.test.fixtures.**date_trigger_dict**(**args*, ***kwargs*)
A cron trigger as a dictionary.

brewtils.test.fixtures.**event_dict**(**args*, ***kwargs*)
An event represented as a dictionary.

brewtils.test.fixtures.**garden_dict**(**args*, ***kwargs*)
A garden as a dictionary.

brewtils.test.fixtures.**instance_dict**(**args*, ***kwargs*)
An instance represented as a dictionary.

brewtils.test.fixtures.**interval_job_dict**(**args*, ***kwargs*)
An interval job represented as a dictionary.

brewtils.test.fixtures.**interval_trigger_dict**(**args*, ***kwargs*)
An interval trigger as a dictionary.

brewtils.test.fixtures.**job_dict**(**args*, ***kwargs*)
A date job represented as a dictionary.

brewtils.test.fixtures.**logging_config_dict**(**args*, ***kwargs*)
A logging config represented as a dictionary.

brewtils.test.fixtures.**nested_parameter_dict**(**args*, ***kwargs*)
Nested Parameter as a dictionary.

brewtils.test.fixtures.**operation_dict**(**args*, ***kwargs*)
An operation as a dictionary.

brewtils.test.fixtures.**parameter_dict**(**args*, ***kwargs*)
Non-nested parameter as a dictionary.

brewtils.test.fixtures.**parent_request**(**args*, ***kwargs*)
A parent request as a model.

brewtils.test.fixtures.**parent_request_dict**(**args*, ***kwargs*)
A parent request represented as a dictionary.

brewtils.test.fixtures.**patch_dict**(**args*, ***kwargs*)
A patch represented as a dictionary.

brewtils.test.fixtures.**patch_dict_no_envelop**(**args*, ***kwargs*)
A patch without an envelope represented as a dictionary.

brewtils.test.fixtures.**patch_dict_no_envelop2**(**args*, ***kwargs*)
A patch without an envelope represented as a dictionary.

brewtils.test.fixtures.**patch_many_dict**(**args*, ***kwargs*)
Multiple patches represented as a dictionary.

brewtils.test.fixtures.**principal_dict**(**args*, ***kwargs*)

brewtils.test.fixtures.**queue_dict**(**args*, ***kwargs*)
A queue represented as a dictionary.

brewtils.test.fixtures.**request_dict**(**args*, ***kwargs*)
A request represented as a dictionary.

brewtils.test.fixtures.**request_file_dict**(**args*, ***kwargs*)
A request file represented as a dictionary.

brewtils.test.fixtures.**request_template_dict**(**args*, ***kwargs*)
Request template as a dictionary.

brewtils.test.fixtures.**role_dict**(**args*, ***kwargs*)

brewtils.test.fixtures.**runner_dict**(**args*, ***kwargs*)
A runner as a dictionary.

brewtils.test.fixtures.**system_dict**(**args*, ***kwargs*)
A system represented as a dictionary.

brewtils.test.fixtures.**system_id**(**args*, ***kwargs*)

brewtils.test.fixtures.**ts_2_dt**(**args*, ***kwargs*)
Feb 2, 2017 as a naive datetime.

brewtils.test.fixtures.**ts_2_dt_utc**(**args*, ***kwargs*)
Feb 2, 2017 UTC as timezone-aware datetime.

brewtils.test.fixtures.**ts_2_epoch**(**args*, ***kwargs*)
Feb 2, 2017 UTC as epoch milliseconds.

brewtils.test.fixtures.**ts_dt**(**args*, ***kwargs*)
Jan 1, 2016 as a naive datetime.

brewtils.test.fixtures.**ts_dt_eastern**(**args*, ***kwargs*)
Jan 1, 2016 US/Eastern as timezone-aware datetime.

brewtils.test.fixtures.**ts_dt_utc**(**args*, ***kwargs*)
Jan 1, 2016 UTC as timezone-aware datetime.

brewtils.test.fixtures.**ts_epoch**(**args*, ***kwargs*)
Jan 1, 2016 UTC as epoch milliseconds.

brewtils.test.fixtures.**ts_epoch_eastern**(**args*, ***kwargs*)
Jan 1, 2016 US/Eastern as epoch milliseconds.

Module contents

5.1.2 Submodules

5.1.3 brewtils.choices module

class brewtils.choices.**FunctionTransformer**
Bases: lark.visitors.Transformer

static arg_pair(*s*)

static func(*s*)

func_args
alias of __builtin__.list

static reference(*s*)

static url(*s*)

url_args
alias of __builtin__.list

brewtils.choices.**parse**(*input_string*, *parse_as=None*)
Attempt to parse a string into a choices dictionary.

Parameters

- **input_string** – The string to parse
- **parse_as** – String specifying how to parse *input_string*. Valid values are 'func' or 'url'. Will try all valid values if None.

Returns Dictionary containing the parse results

Raises lark.common.ParseError – Unable to find a valid parsing of *input_string*

5.1.4 brewtils.decorators module

brewtils.decorators.**system**(*cls=None*, *bg_name=None*, *bg_version=None*)
Class decorator that marks a class as a beer-garden System

Creates some properties on the class:

- _bg_name: an optional system name
- _bg_version: an optional system version
- _bg_commands: holds all registered commands
- _current_request: Reference to the currently executing request

Parameters

- **cls** – The class to decorated
- **bg_name** – Optional plugin name
- **bg_version** – Optional plugin version

Returns The decorated class

brewtils.decorators.**parameter**(*_wrapped=None*, *key=None*, *type=None*, *multi=None*, *display_name=None*, *optional=None*, *default=None*, *description=None*, *choices=None*, *nullable=None*, *maximum=None*, *minimum=None*, *regex=None*, *is_kwarg=None*, *model=None*, *form_input_type=None*)

Decorator that enables Parameter specifications for a beer-garden Command

This is intended to be used when more specification is desired for a Parameter.

For example:

```
@parameter(
    key="message",
    description="Message to echo",
    optional=True,
    type="String",
    default="Hello, World!",
)
def echo(self, message):
    return message
```

Parameters

- **_wrapped** – The function to decorate. This is handled as a positional argument and shouldn't be explicitly set.
- **key** – String specifying the parameter identifier. Must match an argument name of the decorated function.
- **type** – String indicating the type to use for this parameter.
- **multi** – Boolean indicating if this parameter is a multi. See documentation for discussion of what this means.
- **display_name** – String that will be displayed as a label in the user interface.
- **optional** – Boolean indicating if this parameter must be specified.
- **default** – The value this parameter will be assigned if not overridden when creating a request.
- **description** – An additional string that will be displayed in the user interface.
- **choices** – List or dictionary specifying allowed values. See documentation for more information.
- **nullable** – Boolean indicating if this parameter is allowed to be null.
- **maximum** – Integer indicating the maximum value of the parameter.
- **minimum** – Integer indicating the minimum value of the parameter.
- **regex** – String describing a regular expression constraint on the parameter.
- **is_kwarg** – Boolean indicating if this parameter is meant to be part of the decorated function's kwargs.
- **model** – Class to be used as a model for this parameter. Must be a Python type object, not an instance.
- **form_input_type** – Only used for string fields. Changes the form input field (e.g. textarea)

Returns The decorated function

brewtils.decorators.**parameters**(**args*)
Specify multiple Parameter definitions at once

This can be useful for commands which have a large number of complicated parameters but aren't good candidates for a Model.

```
@parameter(**params[cmd1][param1])
@parameter(**params[cmd1][param2])
@parameter(**params[cmd1][param3])
def cmd1(self, **kwargs):
    pass
```

Can become:

```
@parameters(params[cmd1])
def cmd1(self, **kwargs):
    pass
```

Parameters ***args** (*iterable*) – Positional arguments The first (and only) positional argument must be a list containing dictionaries that describe parameters.

Returns The decorated function

Return type func

brewtils.decorators.**command**(*_wrapped=None*, *command_type='ACTION'*, *output_type='STRING'*, *hidden=False*, *schema=None*, *form=None*, *template=None*, *icon_name=None*, *description=None*)
Decorator that marks a function as a beer-garden command

For example:

```
@command(output_type='JSON')
def echo_json(self, message):
    return message
```

Parameters

- **_wrapped** – The function to decorate. This is handled as a positional argument and shouldn't be explicitly set.
- **command_type** – The command type. Valid options are Command.COMMAND_TYPES.
- **output_type** – The output type. Valid options are Command.OUTPUT_TYPES.
- **schema** – A custom schema definition.
- **form** – A custom form definition.
- **template** – A custom template definition.
- **icon_name** – The icon name. Should be either a FontAwesome or a Glyphicon name.
- **hidden** – Status whether command is visible on the user interface.
- **description** – The command description. Will override the function's docstring.

Returns The decorated function

brewtils.decorators.**command_registrar**(*cls=None*, *bg_name=None*, *bg_version=None*)

Class decorator that marks a class as a beer-garden System

Creates some properties on the class:

- `_bg_name`: an optional system name
- `_bg_version`: an optional system version
- `_bg_commands`: holds all registered commands
- `_current_request`: Reference to the currently executing request

Parameters

- **cls** – The class to decorated
- **bg_name** – Optional plugin name
- **bg_version** – Optional plugin version

Returns The decorated class

brewtils.decorators.**plugin_param**(*_wrapped=None*, *key=None*, *type=None*, *multi=None*, *display_name=None*, *optional=None*, *default=None*, *description=None*, *choices=None*, *nullable=None*, *maximum=None*, *minimum=None*, *regex=None*, *is_kwarg=None*, *model=None*, *form_input_type=None*)

Decorator that enables Parameter specifications for a beer-garden Command

This is intended to be used when more specification is desired for a Parameter.

For example:

```
@parameter(
    key="message",
    description="Message to echo",
    optional=True,
    type="String",
    default="Hello, World!",
)
def echo(self, message):
    return message
```

Parameters

- **_wrapped** – The function to decorate. This is handled as a positional argument and shouldn't be explicitly set.
- **key** – String specifying the parameter identifier. Must match an argument name of the decorated function.
- **type** – String indicating the type to use for this parameter.
- **multi** – Boolean indicating if this parameter is a multi. See documentation for discussion of what this means.
- **display_name** – String that will be displayed as a label in the user interface.
- **optional** – Boolean indicating if this parameter must be specified.
- **default** – The value this parameter will be assigned if not overridden when creating a request.

- **description** – An additional string that will be displayed in the user interface.
- **choices** – List or dictionary specifying allowed values. See documentation for more information.
- **nullable** – Boolean indicating if this parameter is allowed to be null.
- **maximum** – Integer indicating the maximum value of the parameter.
- **minimum** – Integer indicating the minimum value of the parameter.
- **regex** – String describing a regular expression constraint on the parameter.
- **is_kwarg** – Boolean indicating if this parameter is meant to be part of the decorated function's kwargs.
- **model** – Class to be used as a model for this parameter. Must be a Python type object, not an instance.
- **form_input_type** – Only used for string fields. Changes the form input field (e.g. textarea)

Returns The decorated function

brewtils.decorators.**register**(*_wrapped=None*, *command_type='ACTION'*, *output_type='STRING'*, *hidden=False*, *schema=None*, *form=None*, *template=None*, *icon_name=None*, *description=None*)

Decorator that marks a function as a beer-garden command

For example:

```
@command(output_type='JSON')
def echo_json(self, message):
    return message
```

Parameters

- **_wrapped** – The function to decorate. This is handled as a positional argument and shouldn't be explicitly set.
- **command_type** – The command type. Valid options are Command.COMMAND_TYPES.
- **output_type** – The output type. Valid options are Command.OUTPUT_TYPES.
- **schema** – A custom schema definition.
- **form** – A custom form definition.
- **template** – A custom template definition.
- **icon_name** – The icon name. Should be either a FontAwesome or a Glyphicon name.
- **hidden** – Status whether command is visible on the user interface.
- **description** – The command description. Will override the function's docstring.

Returns The decorated function

5.1.5 brewtils.errors module

exception brewtils.errors.**AckAndContinueException**

Bases: *brewtils.errors.RequestProcessException*

exception brewtils.errors.**AckAndDieException**
Bases: *brewtils.errors.RequestProcessException*

exception brewtils.errors.**AuthorizationRequired**
Bases: *brewtils.errors.RestClientError*

Error indicating a 401 was raised on the server

brewtils.errors.**BGConflictError**
alias of *brewtils.errors.ConflictError*

exception brewtils.errors.**BGGivesUpError**
Bases: *brewtils.errors.RequestProcessException*

Special exception that indicates Beer-garden is giving up

This exception is not raised directly, instead it's a special value for a request's error_class attribute. It indicates the request may have information that has not been persisted to the database, but Beer-garden is choosing to abandon further attempts to update it.

Typically indicates a request output is too large or the maximum number of update retry attempts has been reached.

brewtils.errors.**BGNotFoundError**
alias of *brewtils.errors.NotFoundError*

brewtils.errors.**BGRequestFailedError**
alias of *brewtils.errors.RequestFailedError*

exception brewtils.errors.**BrewtilsException**
Bases: exceptions.Exception

Base exception

exception brewtils.errors.**ConflictError**
Bases: *brewtils.errors.RestClientError*

Error indicating a 409 was raised on the server

brewtils.errors.**ConnectionTimeoutError**
alias of *brewtils.errors.TimeoutExceededError*

exception brewtils.errors.**DeleteError**
Bases: *brewtils.errors.RestServerError*

Error Indicating a server Error occurred performing a DELETE

exception brewtils.errors.**DiscardMessageException**
Bases: *brewtils.errors.RequestProcessException*

Raising an instance will result in a message not being requeued

exception brewtils.errors.**ErrorLogLevelCritical**
Bases: exceptions.Exception

Mixin to log an exception at the CRITICAL level

exception brewtils.errors.**ErrorLogLevelDebug**
Bases: exceptions.Exception

Mixin to log an exception at the DEBUG level

exception brewtils.errors.**ErrorLogLevelError**
Bases: exceptions.Exception

Mixin to log an exception at the ERROR level

exception brewtils.errors.**ErrorLogLevelInfo**
Bases: exceptions.Exception

Mixin to log an exception at the INFO level

exception brewtils.errors.**ErrorLogLevelWarning**
Bases: exceptions.Exception

Mixin to log an exception at the WARNING level

exception brewtils.errors.**FetchError**
Bases: *brewtils.errors.RestError*

Error Indicating a server Error occurred performing a GET

exception brewtils.errors.**ModelError**
Bases: *brewtils.errors.BrewtilsException*

Base exception for model errors

exception brewtils.errors.**ModelValidationError**
Bases: *brewtils.errors.ModelError*

Invalid model

exception brewtils.errors.**NoAckAndDieException**
Bases: *brewtils.errors.RequestProcessException*

exception brewtils.errors.**NotFoundError**
Bases: *brewtils.errors.RestClientError*

Error Indicating a 404 was raised on the server

exception brewtils.errors.**PluginError**
Bases: *brewtils.errors.BrewtilsException*

Generic error class

exception brewtils.errors.**PluginParamError**
Bases: *brewtils.errors.PluginError*

Error used when plugins have illegal parameters

exception brewtils.errors.**PluginValidationError**
Bases: *brewtils.errors.PluginError*

Plugin could not be validated successfully

exception brewtils.errors.**RepublishRequestException**(*request*, *headers*)
Bases: *brewtils.errors.RequestProcessException*

Republish to the end of the message queue

Parameters

- **request** – The Request to republish
- **headers** – A dictionary of headers to be used by *brewtils.pika.PikaConsumer*

exception brewtils.errors.**RequestFailedError**(*request*)
Bases: *brewtils.errors.RestError*

Request returned with a 200, but the status was ERROR

exception brewtils.errors.**RequestForbidden**
Bases: *brewtils.errors.RestClientError*

Error indicating a 403 was raised on the server

exception brewtils.errors.**RequestProcessException**
Bases: *brewtils.errors.BrewtilsException*

Base for exceptions that occur during request processing

exception brewtils.errors.**RequestProcessingError**
Bases: *brewtils.errors.AckAndContinueException*

exception brewtils.errors.**RequestPublishException**
Bases: *brewtils.errors.BrewtilsException*

Error while publishing request

exception brewtils.errors.**RequestStatusTransitionError**
Bases: *brewtils.errors.ModelValidationError*

A status update was an invalid transition

exception brewtils.errors.**RestClientError**
Bases: *brewtils.errors.RestError*

Wrapper for all 4XX errors

exception brewtils.errors.**RestConnectionError**
Bases: *brewtils.errors.RestServerError*

Error indicating a connection error while performing a request

exception brewtils.errors.**RestError**
Bases: *brewtils.errors.BrewtilsException*

Base exception for REST errors

exception brewtils.errors.**RestServerError**
Bases: *brewtils.errors.RestError*

Wrapper for all 5XX errors

exception brewtils.errors.**SaveError**
Bases: *brewtils.errors.RestServerError*

Error Indicating a server Error occurred performing a POST/PUT

exception brewtils.errors.**SuppressStacktrace**
Bases: exceptions.Exception

Mixin that will suppress stacktrace logging

exception brewtils.errors.**TimeoutExceededError**
Bases: *brewtils.errors.RestClientError*

Error indicating a timeout occurred waiting for a request to complete

exception brewtils.errors.**TooLargeError**
Bases: *brewtils.errors.RestClientError*

Error indicating a 413 was raised on the server

exception brewtils.errors.**ValidationError**
Bases: *brewtils.errors.RestClientError*

Error Indicating a client (400) Error occurred performing a POST/PUT

brewtils.errors.**WaitExceededError**
alias of *brewtils.errors.TimeoutExceededError*

brewtils.errors.**parse_exception_as_json**(*exc*)

Attempt to parse an Exception to a JSON string.

If the exception has a single argument, no attributes, and the attribute can be converted to a valid JSON string, then that will be returned.

Otherwise, a string version of the following form will be returned:

```
{
    "message": "",
    "arguments": [],
    "attributes": {}
}
```

Where "message" is just str(exc), "arguments" is a list of all the arguments passed to the exception attempted to be converted to a valid JSON string, and "attributes" are the attributes of the exception class.

If parsing fails at all, then a simple str() will be applied either the argument or attribute value.

Note: On python version 2, errors with custom attributes do not list those attributes as arguments.

Parameters **exc** (*Exception*) – The exception you would like to format as JSON.

Raises ValueError – If the exception passed in is not an Exception.

Returns A valid JSON string representing (the best we can) the exception.

5.1.6 brewtils.log module

Brewtils Logging Utilities

This module streamlines loading logging configuration from Beergarden.

Example

To use this just call `configure_logging` sometime before you initialize your Plugin object:

```
from brewtils import configure_logging, get_connection_info, Plugin

# Load BG connection info from environment and command line args
connection_info = get_connection_info(sys.argv[1:])

configure_logging(system_name='systemX', **connection_info)

plugin = Plugin(
    my_client,
    name='systemX,
    version='0.0.1',
    **connection_info
)
plugin.run()
```

brewtils.log.**configure_logging**(*raw_config*, *namespace=None*, *system_name=None*, *system_version=None*, *instance_name=None*)

Load and enable a logging configuration from Beergarden

WARNING: This method will modify the current logging configuration.

The configuration will be template substituted using the keyword arguments passed to this function. For example, a handler like this:

```
handlers:
    file:
        backupCount: 5
        class: "logging.handlers.RotatingFileHandler"
        encoding: utf8
        formatter: default
        level: INFO
        maxBytes: 10485760
        filename: "$system_name.log"
```

Will result in logging to a file with the same name as the given system_name.

This will also ensure that directories exist for any file-based handlers. Default behavior for the Python logging module is to not create directories that do not already exist, which would dramatically lower the utility of templating.

Parameters

- **raw_config** – Configuration to apply
- **namespace** – Used for configuration templating
- **system_name** – Used for configuration templating
- **system_version** – Used for configuration templating
- **instance_name** – Used for configuration templating

Returns None

brewtils.log.**convert_logging_config**(*logging_config*)

Transform a LoggingConfig object into a Python logging configuration

Parameters **logging_config** – Beergarden logging config

Returns The logging configuration

Return type dict

brewtils.log.**default_config**(*level='INFO'*)

Get a basic logging configuration with the given level

brewtils.log.**find_log_file**()

Find the file name for the first file handler attached to the root logger

brewtils.log.**get_logging_config**(*system_name=None*, ***kwargs*)

Retrieve a logging configuration from Beergarden

Parameters

- **system_name** – Name of the system to load
- ****kwargs** – Beergarden connection parameters

Returns The logging configuration for the specified system

Return type dict

brewtils.log.**get_python_logging_config**(*bg_host*, *bg_port*, *system_name*, *ca_cert=None*, *client_cert=None*, *ssl_enabled=None*)

DEPRECATED: Get Beergarden's logging configuration

This method is deprecated - consider using *get_logging_config()*

Parameters

- **bg_host** (*str*) – Beergarden host
- **bg_port** (*int*) – Beergarden port
- **system_name** (*str*) – Name of the system
- **ca_cert** (*str*) – Path to CA certificate file
- **client_cert** (*str*) – Path to client certificate file
- **ssl_enabled** (*bool*) – Use SSL when connection to Beergarden

Returns The logging configuration for the specified system

Return type dict

brewtils.log.**read_log_file**(*log_file*, *start_line=None*, *end_line=None*)

Read lines from a log file

Parameters

- **log_file** – The file to read from
- **start_line** – Starting line to read
- **end_line** – Ending line to read

Returns Lines read from the file

brewtils.log.**setup_logger**(*bg_host*, *bg_port*, *system_name*, *ca_cert=None*, *client_cert=None*, *ssl_enabled=None*)

DEPRECATED: Set Python logging to use configuration from Beergarden API

This method is deprecated - consider using *configure_logging()*

This method will overwrite the current logging configuration.

Parameters

- **bg_host** (*str*) – Beergarden host
- **bg_port** (*int*) – Beergarden port
- **system_name** (*str*) – Name of the system
- **ca_cert** (*str*) – Path to CA certificate file
- **client_cert** (*str*) – Path to client certificate file
- **ssl_enabled** (*bool*) – Use SSL when connection to Beergarden

Returns: None

5.1.7 brewtils.models module

class brewtils.models.**BaseModel**

Bases: object

schema = None

class brewtils.models.**System**(*name=None*, *description=None*, *version=None*, *id=None*, *max_instances=None*, *instances=None*, *commands=None*, *icon_name=None*, *display_name=None*, *metadata=None*, *namespace=None*, *local=None*)

Bases: *brewtils.models.BaseModel*

get_command_by_name(*command_name*)
Retrieve a particular command from the system

Parameters **command_name** (*str*) – The command name

Returns The command if it exists, None otherwise

Return type *Command*

get_instance(*name*)
DEPRECATED: Please use get_instance_by_name instead

get_instance_by_id(*id*, *raise_missing=False*)
Get an instance that currently exists in the system

Parameters

- **id** (*str*) – The instance id
- **raise_missing** (*bool*) – If True, raise an exception if an Instance with the
- **id is not found. If False, will return None in that case.** (*given*) –

Returns The instance if it exists, None otherwise

Return type *Instance*

Raises ModelError – Instance was not found and raise_missing=True

get_instance_by_name(*name*, *raise_missing=False*)
Get an instance that currently exists in the system

Parameters

- **name** (*str*) – The instance name
- **raise_missing** (*bool*) – If True, raise an exception if an Instance with the
- **name is not found. If False, will return None in that case.** (*given*) –

Returns The instance if it exists, None otherwise

Return type *Instance*

Raises ModelError – Instance was not found and raise_missing=True

has_different_commands(*commands*)
Check if a set of commands is different than the current commands

Parameters **commands** (*Sequence[Command]*) – Command collection for comparison

Returns True if the given Commands differ, False if they are identical

Return type bool

has_instance(*name*)
Determine if an instance currently exists in the system

Parameters **name** (*str*) – The instance name

Returns True if an instance with the given name exists, False otherwise

Return type bool

instance_names

schema = 'SystemSchema'

class brewtils.models.**Instance**(*name=None*, *description=None*, *id=None*, *status=None*, *status_info=None*, *queue_type=None*, *queue_info=None*, *icon_name=None*, *metadata=None*)

Bases: *brewtils.models.BaseModel*

INSTANCE_STATUSES = set(['DEAD', 'INITIALIZING', 'PAUSED', 'RUNNING', 'STARTING', 'STO

schema = 'InstanceSchema'

class brewtils.models.**Command**(*name=None*, *description=None*, *parameters=None*, *command_type=None*, *output_type=None*, *schema=None*, *form=None*, *template=None*, *icon_name=None*, *hidden=False*)

Bases: *brewtils.models.BaseModel*

COMMAND_TYPES = ('ACTION', 'INFO', 'EPHEMERAL', 'ADMIN')

OUTPUT_TYPES = ('STRING', 'JSON', 'XML', 'HTML', 'JS', 'CSS')

get_parameter_by_key(*key*)

Lookup a Parameter using a given key

Parameters **key** (*str*) – The Parameter key to use

Returns

A Parameter with the given key

If a Parameter with the given key does not exist None will be returned.

Return type *Parameter* (Optional)

has_different_parameters(*parameters*)

Determine if parameters differ from the current parameters

Parameters **parameters** (*Sequence[Parameter]*) – Parameter collection for comparison

Returns True if the given Parameters differ, False if they are identical

Return type bool

parameter_keys()

Get a list of all Parameter keys

Returns A list containing each Parameter's key attribute

Return type list[str]

parameter_keys_by_type(*desired_type*)

Get a list of all Parameter keys, filtered by Parameter type

Parameters **desired_type** (*str*) – Parameter type

Returns A list containing matching Parameters' key attribute

Return type list[str]

schema = 'CommandSchema'

class brewtils.models.**Parameter**(*key*, *type=None*, *multi=None*, *display_name=None*, *optional=None*, *default=None*, *description=None*, *choices=None*, *parameters=None*, *nullable=None*, *maximum=None*, *minimum=None*, *regex=None*, *form_input_type=None*, *type_info=None*)

Bases: *brewtils.models.BaseModel*

FORM_INPUT_TYPES = ('textarea',)

TYPES = ('String', 'Integer', 'Float', 'Boolean', 'Any', 'Dictionary', 'Date', 'DateTi

is_different(*other*)

keys_by_type(*desired_type*)
Gets all keys by the specified type.

Since parameters can be nested, this method will also return all keys of all nested parameters. The return value is a possibly nested list, where the first value of each list is going to be a string, while the next value is a list.

Parameters **desired_type** (*str*) – Desired type

Returns An empty list if the type does not exist, otherwise it will be a list containing at least one entry which is a string, each subsequent entry is a nested list with the same structure.

schema = 'ParameterSchema'

class brewtils.models.**Request**(*system=None*, *system_version=None*, *instance_name=None*, *namespace=None*, *command=None*, *id=None*, *parent=None*, *children=None*, *parameters=None*, *comment=None*, *output=None*, *output_type=None*, *status=None*, *command_type=None*, *created_at=None*, *error_class=None*, *metadata=None*, *updated_at=None*, *has_parent=None*, *requester=None*)
Bases: *brewtils.models.RequestTemplate*

COMMAND_TYPES = ('ACTION', 'INFO', 'EPHEMERAL', 'ADMIN')

COMPLETED_STATUSES = ('CANCELED', 'SUCCESS', 'ERROR')

OUTPUT_TYPES = ('STRING', 'JSON', 'XML', 'HTML', 'JS', 'CSS')

STATUS_LIST = ('CREATED', 'RECEIVED', 'IN_PROGRESS', 'CANCELED', 'SUCCESS', 'ERROR')

classmethod from_template(*template*, ***kwargs*)
Create a Request instance from a RequestTemplate

Parameters

- **template** – The RequestTemplate to use
- ****kwargs** – Optional overrides to use in place of the template's attributes

Returns The new Request instance

is_ephemeral

is_json

schema = 'RequestSchema'

status

class brewtils.models.**PatchOperation**(*operation=None*, *path=None*, *value=None*)
Bases: *brewtils.models.BaseModel*

schema = 'PatchSchema'

class brewtils.models.**Choices**(*type=None*, *display=None*, *value=None*, *strict=None*, *details=None*)
Bases: *brewtils.models.BaseModel*

DISPLAYS = ('select', 'typeahead')

TYPES = ('static', 'url', 'command')

schema = 'ChoicesSchema'

class brewtils.models.**LoggingConfig**(*level=None*, *handlers=None*, *formatters=None*, *loggers=None*)

Bases: *brewtils.models.BaseModel*

DEFAULT_FORMAT = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'

DEFAULT_HANDLER = {'class': 'logging.StreamHandler', 'formatter': 'default', 'stream

LEVELS = ('DEBUG', 'INFO', 'WARN', 'ERROR')

SUPPORTED_HANDLERS = ('stdout', 'file', 'logstash')

formatter_names

get_plugin_log_config(***kwargs*)

Get a specific plugin logging configuration.

It is possible for different systems to have different logging configurations. This method will create the correct plugin logging configuration and return it. If a specific logger is not found for a system, then the current logging configuration will be returned.

Keyword Arguments **information for a system**(*Identifying*) –

Returns The logging configuration for this system

handler_names

schema = 'LoggingConfigSchema'

class brewtils.models.**Event**(*name=None*, *namespace=None*, *garden=None*, *metadata=None*, *timestamp=None*, *payload_type=None*, *payload=None*, *error=None*, *error_message=None*)

Bases: *brewtils.models.BaseModel*

schema = 'EventSchema'

class brewtils.models.**Events**

Bases: enum.Enum

ALL_QUEUES_CLEARED = 15

BARTENDER_STARTED = 3

BARTENDER_STOPPED = 4

BREWVIEW_STARTED = 1

BREWVIEW_STOPPED = 2

DB_CREATE = 16

DB_DELETE = 18

DB_UPDATE = 17

ENTRY_STARTED = 31

ENTRY_STOPPED = 32

FILE_CREATED = 24

GARDEN_CREATED = 19

GARDEN_ERROR = 28

GARDEN_NOT_CONFIGURED = 29

GARDEN_REMOVED = 21

GARDEN_STARTED = 25

GARDEN_STOPPED = 26

GARDEN_SYNC = 30

GARDEN_UNREACHABLE = 27

GARDEN_UPDATED = 20

INSTANCE_INITIALIZED = 8

INSTANCE_STARTED = 9

INSTANCE_STOPPED = 10

INSTANCE_UPDATED = 23

JOB_CREATED = 33

JOB_DELETED = 34

JOB_PAUSED = 35

JOB_RESUMED = 36

PLUGIN_LOGGER_FILE_CHANGE = 37

QUEUE_CLEARED = 14

REQUEST_CANCELED = 29

REQUEST_COMPLETED = 7

REQUEST_CREATED = 5

REQUEST_STARTED = 6

REQUEST_UPDATED = 22

RUNNER_REMOVED = 40

RUNNER_STARTED = 38

RUNNER_STOPPED = 39

SYSTEM_CREATED = 11

SYSTEM_REMOVED = 13

SYSTEM_UPDATED = 12

class brewtils.models.**Queue**(*name=None*, *system=None*, *version=None*, *instance=None*, *system_id=None*, *display=None*, *size=None*)

Bases: *brewtils.models.BaseModel*

schema = 'QueueSchema'

class brewtils.models.**Principal**(*id=None*, *username=None*, *roles=None*, *permissions=None*, *preferences=None*, *metadata=None*)

Bases: *brewtils.models.BaseModel*

schema = 'PrincipalSchema'

class brewtils.models.**Role**(*id=None*, *name=None*, *description=None*, *permissions=None*)

Bases: *brewtils.models.BaseModel*

schema = 'RoleSchema'

class brewtils.models.**RefreshToken**(*id=None*, *issued=None*, *expires=None*, *payload=None*)
Bases: *brewtils.models.BaseModel*

schema = 'RefreshTokenSchema'

class brewtils.models.**Job**(*id=None*, *name=None*, *trigger_type=None*, *trigger=None*, *request_template=None*, *misfire_grace_time=None*, *coalesce=None*, *next_run_time=None*, *success_count=None*, *error_count=None*, *status=None*, *max_instances=None*)
Bases: *brewtils.models.BaseModel*

STATUS_TYPES = set(['PAUSED', 'RUNNING'])

TRIGGER_TYPES = set(['cron', 'date', 'file', 'interval'])

schema = 'JobSchema'

class brewtils.models.**RequestFile**(*storage_type=None*, *filename=None*, *id=None*)
Bases: *brewtils.models.BaseModel*

schema = 'RequestFileSchema'

class brewtils.models.**File**(*id=None*, *owner_id=None*, *owner_type=None*, *updated_at=None*, *file_name=None*, *file_size=None*, *chunks=None*, *chunk_size=None*, *owner=None*)
Bases: *brewtils.models.BaseModel*

schema = 'FileSchema'

class brewtils.models.**FileChunk**(*id=None*, *file_id=None*, *offset=None*, *data=None*, *owner=None*)
Bases: *brewtils.models.BaseModel*

schema = 'FileChunkSchema'

class brewtils.models.**FileStatus**(*owner_id=None*, *owner_type=None*, *updated_at=None*, *file_name=None*, *file_size=None*, *chunks=None*, *chunk_size=None*, *chunk_id=None*, *file_id=None*, *offset=None*, *data=None*, *valid=None*, *missing_chunks=None*, *expected_max_size=None*, *size_ok=None*, *expected_number_of_chunks=None*, *number_of_chunks=None*, *chunks_ok=None*, *operation_complete=None*, *message=None*)
Bases: *brewtils.models.BaseModel*

schema = 'FileStatusSchema'

class brewtils.models.**RequestTemplate**(*system=None*, *system_version=None*, *instance_name=None*, *namespace=None*, *command=None*, *command_type=None*, *parameters=None*, *comment=None*, *metadata=None*, *output_type=None*)
Bases: *brewtils.models.BaseModel*

TEMPLATE_FIELDS = ['system', 'system_version', 'instance_name', 'namespace', 'command'

schema = 'RequestTemplateSchema'

class brewtils.models.**DateTrigger**(*run_date=None*, *timezone=None*)
Bases: *brewtils.models.BaseModel*

scheduler_attributes

scheduler_kwargs

schema = 'DateTriggerSchema'

class brewtils.models.**CronTrigger**(*year=None*, *month=None*, *day=None*, *week=None*, *day_of_week=None*, *hour=None*, *minute=None*, *second=None*, *start_date=None*, *end_date=None*, *timezone=None*, *jitter=None*)

Bases: *brewtils.models.BaseModel*

scheduler_attributes

scheduler_kwargs

schema = 'CronTriggerSchema'

class brewtils.models.**IntervalTrigger**(*weeks=None*, *days=None*, *hours=None*, *minutes=None*, *seconds=None*, *start_date=None*, *end_date=None*, *timezone=None*, *jitter=None*, *reschedule_on_finish=None*)

Bases: *brewtils.models.BaseModel*

scheduler_attributes

scheduler_kwargs

schema = 'IntervalTriggerSchema'

class brewtils.models.**FileTrigger**(*pattern=None*, *path=None*, *recursive=None*, *callbacks=None*)

Bases: *brewtils.models.BaseModel*

scheduler_attributes

scheduler_kwargs

schema = 'FileTriggerSchema'

class brewtils.models.**Garden**(*id=None*, *name=None*, *status=None*, *status_info=None*, *namespaces=None*, *systems=None*, *connection_type=None*, *connection_params=None*)

Bases: *brewtils.models.BaseModel*

GARDEN_STATUSES = set(['BLOCKED', 'ERROR', 'INITIALIZING', 'NOT_CONFIGURED', 'RUNNING'

schema = 'GardenSchema'

class brewtils.models.**Operation**(*model=None*, *model_type=None*, *args=None*, *kwargs=None*, *target_garden_name=None*, *source_garden_name=None*, *operation_type=None*)

Bases: *brewtils.models.BaseModel*

schema = 'OperationSchema'

5.1.8 brewtils.plugin module

class brewtils.plugin.**Plugin**(*client=None*, *system=None*, *logger=None*, ***kwargs*)

Bases: `object`

A Beer-garden Plugin

This class represents a Beer-garden Plugin - a continuously-running process that can receive and process Requests.

To work, a Plugin needs a Client instance - an instance of a class defining which Requests this plugin can accept and process. The easiest way to define a `Client` is by annotating a class with the `@system` decorator.

A Plugin needs certain pieces of information in order to function correctly. These can be grouped into two high-level categories: identifying information and connection information.

Identifying information is how Beer-garden differentiates this Plugin from all other Plugins. If you already have fully-defined System model you can pass that directly to the Plugin (`system=my_system`). However, normally it's simpler to pass the pieces directly:

- `name` (required)
- `version` (required)
- `instance_name` (required, but defaults to "default")
- `namespace`
- `description`
- `icon_name`
- `metadata`
- `display_name`

Connection information tells the Plugin how to communicate with Beer-garden. The most important of these is the `bg_host` (to tell the plugin where to find the Beer-garden you want to connect to):

- `bg_host`
- `bg_port`
- `bg_url_prefix`
- `ssl_enabled`
- `ca_cert`
- `ca_verify`
- `client_cert`

An example plugin might look like this:

```
Plugin(
    name="Test",
    version="1.0.0",
    instance_name="default",
    namespace="test plugins",
    description="A Test",
    bg_host="localhost",
)
```

Plugins use Yapconf for configuration loading, which means that values can be discovered from sources other than direct argument passing. Config can be passed as command line arguments:

```
python my_plugin.py --bg-host localhost
```

Values can also be specified as environment variables with a "BG_" prefix:

```
BG_HOST=localhost python my_plugin.py
```

Plugins service requests using a `concurrent.futures.ThreadPoolExecutor`. The maximum number of threads available is controlled by the `max_concurrent` argument.

Warning: Normally the processing of each Request occurs in a distinct thread context. If you need to access shared state please be careful to use appropriate concurrency mechanisms.

Warning: The default value for `max_concurrent` is 5, but setting it to 1 is allowed. This means that a Plugin will essentially be single-threaded, but realize this means that if the Plugin invokes a Command on itself in the course of processing a Request then the Plugin **will** deadlock!

Parameters

- **client** – Instance of a class annotated with `@system`.
- **bg_host** (*str*) – Beer-garden hostname
- **bg_port** (*int*) – Beer-garden port
- **bg_url_prefix** (*str*) – URL path that will be used as a prefix when communicating with Beer-garden. Useful if Beer-garden is running on a URL other than '/'.
- **ssl_enabled** (*bool*) – Whether to use SSL for Beer-garden communication
- **ca_cert** (*str*) – Path to certificate file containing the certificate of the authority that issued the Beer-garden server certificate
- **ca_verify** (*bool*) – Whether to verify Beer-garden server certificate
- **client_cert** (*str*) – Path to client certificate to use when communicating with Beer-garden
- **api_version** (*int*) – Beer-garden API version to use
- **client_timeout** (*int*) – Max time to wait for Beer-garden server response
- **username** (*str*) – Username for Beer-garden authentication
- **password** (*str*) – Password for Beer-garden authentication
- **access_token** (*str*) – Access token for Beer-garden authentication
- **refresh_token** (*str*) – Refresh token for Beer-garden authentication
- **system** (*brewtils.models.System*) – A Beer-garden System definition. Incompatible with name, version, description, display_name, icon_name, max_instances, and metadata parameters.
- **name** (*str*) – System name
- **version** (*str*) – System version
- **description** (*str*) – System description
- **display_name** (*str*) – System display name
- **icon_name** (*str*) – System icon name
- **max_instances** (*int*) – System maximum instances
- **metadata** (*dict*) – System metadata
- **instance_name** (*str*) – Instance name
- **namespace** (*str*) – Namespace name

- **logger** (logging.Logger) – Logger that will be used by the Plugin. Passing a logger will prevent the Plugin from preforming any additional logging configuration.
- **worker_shutdown_timeout** (*int*) – Time to wait during shutdown to finish processing
- **max_concurrent** (*int*) – Maximum number of requests to process concurrently
- **max_attempts** (*int*) – Number of times to attempt updating of a Request before giving up. Negative numbers are interpreted as no maximum.
- **max_timeout** (*int*) – Maximum amount of time to wait between Request update attempts. Negative numbers are interpreted as no maximum.
- **starting_timeout** (*int*) – Initial time to wait between Request update attempts. Will double on subsequent attempts until reaching max_timeout.
- **mq_max_attempts** (*int*) – Number of times to attempt reconnection to message queue before giving up. Negative numbers are interpreted as no maximum.
- **mq_max_timeout** (*int*) – Maximum amount of time to wait between message queue reconnect attempts. Negative numbers are interpreted as no maximum.
- **mq_starting_timeout** (*int*) – Initial time to wait between message queue reconnect attempts. Will double on subsequent attempts until reaching mq_max_timeout.
- **working_directory** (*str*) – Path to a preferred working directory. Only used when working with bytes parameters.

bg_host

bg_port

bg_url_prefix

bm_client

ca_cert

ca_verify

client

client_cert

connection_parameters

instance

instance_name

logger

max_attempts

max_concurrent

max_timeout

metadata

run()

shutdown_event

ssl_enabled

starting_timeout

system

unique_name

class brewtils.plugin.**PluginBase**(**args*, ***kwargs*)
Bases: *brewtils.plugin.Plugin*

class brewtils.plugin.**RemotePlugin**(**args*, ***kwargs*)
Bases: *brewtils.plugin.Plugin*

5.1.9 brewtils.queues module

This module currently exists to maintain backwards compatibility.

class brewtils.queues.**PikaClient**(*host='localhost'*, *port=5672*, *user='guest'*, *password='guest'*, *connection_attempts=3*, *heartbeat_interval=3600*, *virtual_host='/'*, *exchange='beer_garden'*, *ssl=None*, *blocked_connection_timeout=None*, ***kwargs*)
Bases: object

Base class for connecting to RabbitMQ using Pika

Parameters

- **host** – RabbitMQ host
- **port** – RabbitMQ port
- **user** – RabbitMQ user
- **password** – RabbitMQ password
- **connection_attempts** – Maximum number of retry attempts
- **heartbeat** – Time between RabbitMQ heartbeats
- **heartbeat_interval** – DEPRECATED, use heartbeat
- **virtual_host** – RabbitMQ virtual host
- **exchange** – Default exchange that will be used
- **ssl** – SSL Options
- **blocked_connection_timeout** – If not None, the value is a non-negative timeout, in seconds, for the connection to remain blocked (triggered by Connection.Blocked from broker); if the timeout expires before connection becomes unblocked, the connection will be torn down, triggering the adapter-specific mechanism for informing client app about the closed connection (e.g., on_close_callback or ConnectionClosed exception) with *reason_code* of *InternalCloseReasons.BLOCKED_CONNECTION_TIMEOUT*.

connection_parameters(***kwargs*)
Get ConnectionParameters associated with this client

Will construct a ConnectionParameters object using parameters passed at initialization as defaults. Any parameters passed in kwargs will override initialization parameters.

Parameters ****kwargs** – Overrides for specific parameters

Returns ConnectionParameters object

Return type pika.ConnectionParameters

connection_url
Connection URL for this client's connection information

Type str

5.1.10 brewtils.request_handling module

class brewtils.request_handling.**AdminProcessor**(*target*, *updater*, *consumer*, *validation_funcs=None*, *logger=None*, *plugin_name=None*, *max_workers=None*, *resolvers=None*, *working_directory=None*)

Bases: *brewtils.request_handling.RequestProcessor*

RequestProcessor with slightly modified process method

process_message(*target*, *request*, *headers*)
Process a message. Intended to be run on an Executor.

Will invoke the command and set the final status / output / error_class.

Will NOT set the status to IN_PROGRESS or set the request context.

Parameters

- **target** – The object to invoke received commands on
- **request** – The parsed Request
- **headers** – Dictionary of headers from the *PikaConsumer*

Returns None

class brewtils.request_handling.**HTTPRequestUpdater**(*ez_client*, *shutdown_event*, ***kwargs*)

Bases: *brewtils.request_handling.RequestUpdater*

RequestUpdater implementation based around an EasyClient.

Parameters

- **ez_client** – EasyClient to use for communication
- **shutdown_event** – *threading.Event* to allow for timely shutdowns
- **logger** – A logger

Keyword Arguments

- **logger** – A logger
- **max_attempts** – Max number of unsuccessful updates before discarding the message
- **max_timeout** – Maximum amount of time (seconds) to wait between update attempts
- **starting_timeout** – Starting time to wait (seconds) between update attempts

shutdown()

update_request(*request*, *headers*)
Sends a Request update to beer-garden

Ephemeral requests do not get updated, so we simply skip them.

If brew-view appears to be down, it will wait for brew-view to come back up before updating.

If this is the final attempt to update, we will attempt a known, good request to give some information to the user. If this attempt fails then we simply discard the message.

Parameters

- **request** – The request to update
- **headers** – A dictionary of headers from the *PikaConsumer*

Returns None

Raises `RepublishMessageException` – The Request update failed (any reason)

class brewtils.request_handling.**NoopUpdater**(**args*, ***kwargs*)
Bases: *brewtils.request_handling.RequestUpdater*

RequestUpdater implementation that explicitly does not update.

shutdown()

update_request(*request*, *headers*)

class brewtils.request_handling.**RequestConsumer**(**args*, ***kwargs*)
Bases: `threading.Thread`

Base class for consumers

Classes deriving from this are expected to provide a concrete implementation for a specific queue type.

After the consumer is created it will be passed to a `RequestProcessor`. The processor will then set the `on_message_callback` property of the consumer to the correct method.

This means when the consumer receives a message it should invoke its own `_on_message_callback` method with the message body and headers as parameters:

```
self._on_message_callback(body, properties.headers)
```

static create(*connection_type=None*, ***kwargs*)
Factory method for consumer creation

Currently the only supported connection_type is "rabbitmq", which will return an instance of `brewtils.pika.PikaConsumer`.

Parameters

- **connection_type** (*str*) – String describing connection type
- **kwargs** – Keyword arguments to be passed to the Consumer initializer

Returns Concrete instance of RequestConsumer

Raises `ValueError` – The specified connection_type does not map to a consumer class

on_message_callback

stop()

stop_consuming()

class brewtils.request_handling.**RequestProcessor**(*target*, *updater*, *consumer*, *validation_funcs=None*, *logger=None*, *plugin_name=None*, *max_workers=None*, *resolvers=None*, *working_directory=None*)
Bases: `object`

Class responsible for coordinating Request processing

The RequestProcessor is responsible for the following: - Defining on_message_received callback that will be invoked by the PikaConsumer - Parsing the request - Invoking the command on the target - Formatting the output - Reporting request updates to Beergarden (using a RequestUpdater)

Parameters

- **target** – Incoming requests will be invoked on this object
- **updater** – RequestUpdater that will be used for updating requests
- **validation_funcs** – List of functions that will called before invoking a command
- **logger** – A logger
- **plugin_name** – The Plugin's unique name
- **max_workers** – Max number of threads to use in the executor pool

on_message_received(*message*, *headers*)
Callback function that will be invoked for received messages

This will attempt to parse the message and then run the parsed Request through all validation functions that this RequestProcessor knows about.

If the request parses cleanly and passes validation it will be submitted to this RequestProcessor's ThreadPoolExecutor for processing.

Parameters

- **message** – The message string
- **headers** – The header dictionary

Returns A future that will complete when processing finishes

Raises

- `DiscardMessageException` – The request failed to parse correctly
- `RequestProcessException` – Validation failures should raise a subclass of this

process_message(*target*, *request*, *headers*)
Process a message. Intended to be run on an Executor.

Will set the status to IN_PROGRESS, invoke the command, and set the final status / output / error_class.

Parameters

- **target** – The object to invoke received commands on
- **request** – The parsed Request
- **headers** – Dictionary of headers from the *PikaConsumer*

Returns None

shutdown()
Stop the RequestProcessor

startup()
Start the RequestProcessor

class brewtils.request_handling.**RequestUpdater**
Bases: `object`

shutdown()

update_request(*request*, *headers*)

5.1.11 brewtils.schema_parser module

class brewtils.schema_parser.**SchemaParser**
Bases: object

Serialize and deserialize Brewtils models

logger = <logging.Logger object>

classmethod parse(*data*, *model_class*, *from_string=False*, ***kwargs*)
Convert a JSON string or dictionary into a model object

Parameters

- **data** – The raw input
- **model_class** – Class object of the desired model type
- **from_string** – True if input is a JSON string, False if a dictionary
- ****kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns A model object

classmethod parse_command(*command*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a command model object

Parameters

- **command** – The raw input
- **from_string** – True if input is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns A Command object

classmethod parse_event(*event*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to an event model object

Parameters

- **event** – The raw input
- **from_string** – True if input is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns An Event object

classmethod parse_garden(*garden*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a garden model object

Parameters

- **garden** – The raw input
- **from_string** – True if input is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns A Garden object

classmethod parse_instance(*instance*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to an instance model object

Parameters

- **instance** – The raw input
- **from_string** – True if input is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns An Instance object

classmethod parse_job(*job*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a job model object

Parameters

- **job** – Raw input
- **from_string** – True if input is a JSON string, False if a dictionary
- ****kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns A Job object.

classmethod parse_logging_config(*logging_config*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a logging config model object

Parameters

- **logging_config** – The raw input
- **from_string** – True if 'input is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns A LoggingConfig object

classmethod parse_operation(*operation*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a garden model object

Parameters

- **operation** – The raw input
- **from_string** – True if input is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns An Operation object

classmethod parse_parameter(*parameter*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a parameter model object

Parameters

- **parameter** – The raw input
- **from_string** – True if input is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns An Parameter object

classmethod parse_patch(*patch*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a patch model object

Note: for our patches, many is _always_ set to True. We will always return a list from this method.

Parameters

- **patch** – The raw input

- **from_string** – True if input is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns A PatchOperation object

classmethod parse_principal(*principal*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a principal model object

Parameters

- **principal** – The raw input
- **from_string** – True if input is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns A Principal object

classmethod parse_queue(*queue*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a queue model object

Parameters

- **queue** – The raw input
- **from_string** – True if input is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns A Queue object

classmethod parse_refresh_token(*refresh_token*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a refresh token object

Parameters

- **refresh_token** – The raw input
- **from_string** – True if input is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns A RefreshToken object

classmethod parse_request(*request*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a request model object

Parameters

- **request** – The raw input
- **from_string** – True if input is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns A Request object

classmethod parse_request_file(*request_file*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a request file model object

Parameters

- **request_file** – The raw input
- **from_string** – True if input is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns A RequestFile object

classmethod parse_role(*role*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a role model object

Parameters

- **role** – The raw input
- **from_string** – True if input is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns A Role object

classmethod parse_runner(*runner*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a runner model object

Parameters

- **runner** – The raw input
- **from_string** – True if input is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns A Runner object

classmethod parse_system(*system*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a system model object

Parameters

- **system** – The raw input
- **from_string** – True if input is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns A System object

classmethod serialize(*model*, *to_string=False*, *schema_name=None*, ***kwargs*)
Convert a model object or list of models into a dictionary or JSON string.

This is potentially recursive - here's how this should work:

- Determine the correct schema to use for serializing. This can be explicitly

passed as an argument, or it can be determined by inspecting the model to serialize. - Determine if the model to serialize is a collection or a single object.

- If it's a single object, serialize it and return that.
- If it's a collection, construct a list by calling this method for each

individual item in the collection. Then serialize **that** and return it.

Parameters

- **model** – The model or model list
- **to_string** – True to generate a JSON string, False to generate a dictionary
- **schema_name** – Name of schema to use for serializing. If None, will be
- **by inspecting model** (*determined*) –
- ****kwargs** – Additional parameters to be passed to the Schema. Note that the 'many' parameter will be set correctly automatically.

Returns A serialized model representation

classmethod serialize_command(*command*, *to_string=True*, ***kwargs*)
Convert a command model into serialized form

Parameters

- **command** – The command object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns Serialized representation of command

classmethod serialize_event(*event*, *to_string=True*, ***kwargs*)
Convert a logging config model into serialized form

Parameters

- **event** – The event object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns Serialized representation of event

classmethod serialize_garden(*garden*, *to_string=True*, ***kwargs*)
Convert an garden model into serialized form

Parameters

- **garden** – The instance object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns Serialized representation of garden

classmethod serialize_instance(*instance*, *to_string=True*, ***kwargs*)
Convert an instance model into serialized form

Parameters

- **instance** – The instance object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns Serialized representation of instance

classmethod serialize_job(*job*, *to_string=True*, ***kwargs*)
Convert a job model into serialized form.

Parameters

- **job** – The job object(s) to be serialized.
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary.
- ****kwargs** – Additional parameters to be passed to the shcema (e.g. many=True)

Returns Serialize representation of job.

classmethod serialize_logging_config(*logging_config*, *to_string=True*, ***kwargs*)
Convert a logging config model into serialize form

Parameters

- **logging_config** – The logging config object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns Serialized representation of logging config

classmethod serialize_operation(*operation*, *to_string=True*, ***kwargs*)
Convert an operation model into serialized form

Parameters

- **operation** – The instance object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns Serialized representation of operation

classmethod serialize_parameter(*parameter*, *to_string=True*, ***kwargs*)
Convert a parameter model into serialized form

Parameters

- **parameter** – The parameter object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns Serialized representation of parameter

classmethod serialize_patch(*patch*, *to_string=True*, ***kwargs*)
Convert a patch model into serialized form

Parameters

- **patch** – The patch object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns Serialized representation of patch

classmethod serialize_principal(*principal*, *to_string=True*, ***kwargs*)
Convert a principal model into serialized form

Parameters

- **principal** – The principal object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns Serialized representation

classmethod serialize_queue(*queue*, *to_string=True*, ***kwargs*)
Convert a queue model into serialized form

Parameters

- **queue** – The queue object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns Serialized representation of queue

classmethod serialize_refresh_token(*refresh_token*, *to_string=True*, ***kwargs*)
Convert a role model into serialized form

Parameters

- **refresh_token** – The token object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns Serialized representation

classmethod serialize_request(*request*, *to_string=True*, ***kwargs*)
Convert a request model into serialized form

Parameters

- **request** – The request object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns Serialized representation of request

classmethod serialize_request_file(*request_file*, *to_string=True*, ***kwargs*)
Convert a request file model into serialized form

Parameters

- **request_file** – The request file object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns Serialized representation of request file

classmethod serialize_role(*role*, *to_string=True*, ***kwargs*)
Convert a role model into serialized form

Parameters

- **role** – The role object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns Serialized representation

classmethod serialize_runner(*runner*, *to_string=True*, ***kwargs*)
Convert a runner model into serialized form

Parameters

- **operation** – The instance object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns Serialized representation of runner

classmethod serialize_system(*system*, *to_string=True*, *include_commands=True*, ***kwargs*)
Convert a system model into serialized form

Parameters

- **system** – The system object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **include_commands** – True if the system's command list should be included
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

Returns Serialized representation of system

5.1.12 brewtils.schemas module

class brewtils.schemas.**SystemSchema**(*strict=True*, ***kwargs*)
Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**InstanceSchema**(*strict=True*, ***kwargs*)
Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**CommandSchema**(*strict=True*, ***kwargs*)
Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**ParameterSchema**(*strict=True*, ***kwargs*)
Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**RequestSchema**(*strict=True*, ***kwargs*)
Bases: brewtils.schemas.RequestTemplateSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**RequestFileSchema**(*strict=True*, ***kwargs*)
Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**FileSchema**(*strict=True*, ***kwargs*)
Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**FileChunkSchema**(*strict=True*, ***kwargs*)
Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**FileStatusSchema**(*strict=True*, ***kwargs*)
Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**PatchSchema**(*strict=True*, ***kwargs*)
Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

unwrap_envelope(*data*, *many*)

Helper function for parsing the different patch formats.

This exists because previously multiple patches serialized like:

```
{
    "operations": [
        {"operation": "replace", ...},
        {"operation": "replace", ...}
        ...
    ]
}
```

But we also wanted to be able to handle a simple list:

```
[
    {"operation": "replace", ...},
    {"operation": "replace", ...}
    ...
]
```

Patches are now (as of v3) serialized as the latter. Prior to v3 they were serialized as the former.

class brewtils.schemas.**LoggingConfigSchema**(*strict=True*, ***kwargs*)

Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**EventSchema**(*strict=True*, ***kwargs*)

Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**QueueSchema**(*strict=True*, ***kwargs*)

Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**PrincipalSchema**(*strict=True*, ***kwargs*)

Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**RoleSchema**(*strict=True*, ***kwargs*)

Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**RefreshTokenSchema**(*strict=True*, ***kwargs*)

Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**JobSchema**(*strict=True*, ***kwargs*)

Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**DateTriggerSchema**(*strict=True*, ***kwargs*)

Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**IntervalTriggerSchema**(*strict=True*, ***kwargs*)
Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**CronTriggerSchema**(*strict=True*, ***kwargs*)
Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**FileTriggerSchema**(*strict=True*, ***kwargs*)
Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**GardenSchema**(*strict=True*, ***kwargs*)
Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

class brewtils.schemas.**OperationSchema**(*strict=True*, ***kwargs*)
Bases: brewtils.schemas.BaseSchema

opts = <marshmallow.schema.SchemaOpts object>

5.1.13 brewtils.specification module

5.1.14 brewtils.stoppable_thread module

class brewtils.stoppable_thread.**StoppableThread**(***kwargs*)
Bases: threading.Thread

Thread class with a stop() method. The thread itself has to check regularly for the stopped() condition.

stop()
Sets the stop event

stopped()
Determines if stop has been called yet.

wait(*timeout=None*)
Delegate wait call to threading.Event

5.1.15 Module contents

brewtils.**command**(*_wrapped=None*, *command_type='ACTION'*, *output_type='STRING'*, *hidden=False*, *schema=None*, *form=None*, *template=None*, *icon_name=None*, *description=None*)
Decorator that marks a function as a beer-garden command

For example:

```
@command(output_type='JSON')
def echo_json(self, message):
    return message
```

Parameters

- **_wrapped** – The function to decorate. This is handled as a positional argument and shouldn't be explicitly set.

- **command_type** – The command type. Valid options are Command.COMMAND_TYPES.
- **output_type** – The output type. Valid options are Command.OUTPUT_TYPES.
- **schema** – A custom schema definition.
- **form** – A custom form definition.
- **template** – A custom template definition.
- **icon_name** – The icon name. Should be either a FontAwesome or a Glyphicon name.
- **hidden** – Status whether command is visible on the user interface.
- **description** – The command description. Will override the function's docstring.

Returns The decorated function

brewtils.**parameter**(*_wrapped=None*, *key=None*, *type=None*, *multi=None*, *display_name=None*, *optional=None*, *default=None*, *description=None*, *choices=None*, *nullable=None*, *maximum=None*, *minimum=None*, *regex=None*, *is_kwarg=None*, *model=None*, *form_input_type=None*)

Decorator that enables Parameter specifications for a beer-garden Command

This is intended to be used when more specification is desired for a Parameter.

For example:

```
@parameter(
    key="message",
    description="Message to echo",
    optional=True,
    type="String",
    default="Hello, World!",
)
def echo(self, message):
    return message
```

Parameters

- **_wrapped** – The function to decorate. This is handled as a positional argument and shouldn't be explicitly set.
- **key** – String specifying the parameter identifier. Must match an argument name of the decorated function.
- **type** – String indicating the type to use for this parameter.
- **multi** – Boolean indicating if this parameter is a multi. See documentation for discussion of what this means.
- **display_name** – String that will be displayed as a label in the user interface.
- **optional** – Boolean indicating if this parameter must be specified.
- **default** – The value this parameter will be assigned if not overridden when creating a request.
- **description** – An additional string that will be displayed in the user interface.
- **choices** – List or dictionary specifying allowed values. See documentation for more information.
- **nullable** – Boolean indicating if this parameter is allowed to be null.

- **maximum** – Integer indicating the maximum value of the parameter.
- **minimum** – Integer indicating the minimum value of the parameter.
- **regex** – String describing a regular expression constraint on the parameter.
- **is_kwarg** – Boolean indicating if this parameter is meant to be part of the decorated function's kwargs.
- **model** – Class to be used as a model for this parameter. Must be a Python type object, not an instance.
- **form_input_type** – Only used for string fields. Changes the form input field (e.g. textarea)

Returns The decorated function

brewtils.**system**(*cls=None*, *bg_name=None*, *bg_version=None*)

Class decorator that marks a class as a beer-garden System

Creates some properties on the class:

- `_bg_name`: an optional system name
- `_bg_version`: an optional system version
- `_bg_commands`: holds all registered commands
- `_current_request`: Reference to the currently executing request

Parameters

- **cls** – The class to decorated
- **bg_name** – Optional plugin name
- **bg_version** – Optional plugin version

Returns The decorated class

class brewtils.**Plugin**(*client=None*, *system=None*, *logger=None*, ***kwargs*)

Bases: `object`

A Beer-garden Plugin

This class represents a Beer-garden Plugin - a continuously-running process that can receive and process Requests.

To work, a Plugin needs a Client instance - an instance of a class defining which Requests this plugin can accept and process. The easiest way to define a `Client` is by annotating a class with the `@system` decorator.

A Plugin needs certain pieces of information in order to function correctly. These can be grouped into two high-level categories: identifying information and connection information.

Identifying information is how Beer-garden differentiates this Plugin from all other Plugins. If you already have fully-defined System model you can pass that directly to the Plugin (`system=my_system`). However, normally it's simpler to pass the pieces directly:

- `name` (required)
- `version` (required)
- `instance_name` (required, but defaults to "default")
- `namespace`

- description
- icon_name
- metadata
- display_name

Connection information tells the Plugin how to communicate with Beer-garden. The most important of these is the bg_host (to tell the plugin where to find the Beer-garden you want to connect to):

- bg_host
- bg_port
- bg_url_prefix
- ssl_enabled
- ca_cert
- ca_verify
- client_cert

An example plugin might look like this:

```
Plugin(
    name="Test",
    version="1.0.0",
    instance_name="default",
    namespace="test plugins",
    description="A Test",
    bg_host="localhost",
)
```

Plugins use Yapconf for configuration loading, which means that values can be discovered from sources other than direct argument passing. Config can be passed as command line arguments:

```
python my_plugin.py --bg-host localhost
```

Values can also be specified as environment variables with a "BG_" prefix:

```
BG_HOST=localhost python my_plugin.py
```

Plugins service requests using a concurrent.futures.ThreadPoolExecutor. The maximum number of threads available is controlled by the max_concurrent argument.

Warning: Normally the processing of each Request occurs in a distinct thread context. If you need to access shared state please be careful to use appropriate concurrency mechanisms.

Warning: The default value for max_concurrent is 5, but setting it to 1 is allowed. This means that a Plugin will essentially be single-threaded, but realize this means that if the Plugin invokes a Command on itself in the course of processing a Request then the Plugin **will** deadlock!

Parameters

- **client** – Instance of a class annotated with @system.

- **bg_host** (*str*) – Beer-garden hostname
- **bg_port** (*int*) – Beer-garden port
- **bg_url_prefix** (*str*) – URL path that will be used as a prefix when communicating with Beer-garden. Useful if Beer-garden is running on a URL other than '/'.
- **ssl_enabled** (*bool*) – Whether to use SSL for Beer-garden communication
- **ca_cert** (*str*) – Path to certificate file containing the certificate of the authority that issued the Beer-garden server certificate
- **ca_verify** (*bool*) – Whether to verify Beer-garden server certificate
- **client_cert** (*str*) – Path to client certificate to use when communicating with Beer-garden
- **api_version** (*int*) – Beer-garden API version to use
- **client_timeout** (*int*) – Max time to wait for Beer-garden server response
- **username** (*str*) – Username for Beer-garden authentication
- **password** (*str*) – Password for Beer-garden authentication
- **access_token** (*str*) – Access token for Beer-garden authentication
- **refresh_token** (*str*) – Refresh token for Beer-garden authentication
- **system** (*brewtils.models.System*) – A Beer-garden System definition. Incompatible with name, version, description, display_name, icon_name, max_instances, and metadata parameters.
- **name** (*str*) – System name
- **version** (*str*) – System version
- **description** (*str*) – System description
- **display_name** (*str*) – System display name
- **icon_name** (*str*) – System icon name
- **max_instances** (*int*) – System maximum instances
- **metadata** (*dict*) – System metadata
- **instance_name** (*str*) – Instance name
- **namespace** (*str*) – Namespace name
- **logger** (`logging.Logger`) – Logger that will be used by the Plugin. Passing a logger will prevent the Plugin from preforming any additional logging configuration.
- **worker_shutdown_timeout** (*int*) – Time to wait during shutdown to finish processing
- **max_concurrent** (*int*) – Maximum number of requests to process concurrently
- **max_attempts** (*int*) – Number of times to attempt updating of a Request before giving up. Negative numbers are interpreted as no maximum.
- **max_timeout** (*int*) – Maximum amount of time to wait between Request update attempts. Negative numbers are interpreted as no maximum.
- **starting_timeout** (*int*) – Initial time to wait between Request update attempts. Will double on subsequent attempts until reaching max_timeout.

- **mq_max_attempts** (*int*) – Number of times to attempt reconnection to message queue before giving up. Negative numbers are interpreted as no maximum.
- **mq_max_timeout** (*int*) – Maximum amount of time to wait between message queue reconnect attempts. Negative numbers are interpreted as no maximum.
- **mq_starting_timeout** (*int*) – Initial time to wait between message queue reconnect attempts. Will double on subsequent attempts until reaching mq_max_timeout.
- **working_directory** (*str*) – Path to a preferred working directory. Only used when working with bytes parameters.

bg_host

bg_port

bg_url_prefix

bm_client

ca_cert

ca_verify

client

client_cert

connection_parameters

instance

instance_name

logger

max_attempts

max_concurrent

max_timeout

metadata

run()

shutdown_event

ssl_enabled

starting_timeout

system

unique_name

class brewtils.**EasyClient**(***kwargs*)

Bases: object

Client for simplified communication with Beergarden

This class is intended to be a middle ground between the RestClient and SystemClient. It provides a 'cleaner' interface to some common Beergarden operations than is exposed by the lower-level RestClient. On the other hand, the SystemClient is much better for generating Beergarden Requests.

Parameters

- **bg_host** (*str*) – Beer-garden hostname

- **bg_port** (*int*) – Beer-garden port
- **bg_url_prefix** (*str*) – URL path that will be used as a prefix when communicating with Beer-garden. Useful if Beer-garden is running on a URL other than '/'.
- **ssl_enabled** (*bool*) – Whether to use SSL for Beer-garden communication
- **ca_cert** (*str*) – Path to certificate file containing the certificate of the authority that issued the Beer-garden server certificate
- **ca_verify** (*bool*) – Whether to verify Beer-garden server certificate
- **client_cert** (*str*) – Path to client certificate to use when communicating with Beer-garden
- **api_version** (*int*) – Beer-garden API version to use
- **client_timeout** (*int*) – Max time to wait for Beer-garden server response
- **username** (*str*) – Username for Beer-garden authentication
- **password** (*str*) – Password for Beer-garden authentication
- **access_token** (*str*) – Access token for Beer-garden authentication
- **refresh_token** (*str*) – Refresh token for Beer-garden authentication

can_connect(***kwargs*)
Determine if the Beergarden server is responding.

Kwargs: Arguments passed to the underlying Requests method

Returns A bool indicating if the connection attempt was successful. Will return False only if a ConnectionError is raised during the attempt. Any other exception will be re-raised.

Raises `requests.exceptions.RequestException` – The connection attempt resulted in an exception that indicates something other than a basic connection error. For example, an error with certificate verification.

clear_all_queues()
Cancel and remove all Requests in all queues

Returns True if the clear was successful

Return type bool

clear_queue(*queue_name*)
Cancel and remove all Requests from a message queue

Parameters **queue_name** (*str*) – The name of the queue to clear

Returns True if the clear was successful

Return type bool

create_job(*job*)
Create a new Job

Parameters **job** (Job) – New Job definition

Returns The newly-created Job

Return type *Job*

create_request(*request*, ***kwargs*)
Create a new Request

Parameters

- **request** – New request definition
- **kwargs** – Extra request parameters

Keyword Arguments

- **blocking** (*bool*) – Wait for request to complete before returning
- **timeout** (*int*) – Maximum seconds to wait for completion

Returns The newly-created Request

Return type *Request*

create_system(*system*)
Create a new System

Parameters **system** (System) – The System to create

Returns The newly-created system

Return type *System*

delete_file(*file_id*)
Delete a given file on the Beer Garden server.

Parameters **file_id** – The beer garden-assigned file id.

Returns The API response

download_file(*file_id*)
Download a file from the Beer Garden server.

Parameters **file_id** – The beer garden-assigned file id.

Returns A file object

find_jobs(***kwargs*)
Find Jobs using keyword arguments as search parameters

Parameters ****kwargs** – Search parameters

Returns List of Jobs matching the search parameters

Return type List[*Job*]

find_requests(***kwargs*)
Find Requests using keyword arguments as search parameters

Parameters ****kwargs** – Search parameters

Returns List of Systems matching the search parameters

Return type List[*Request*]

find_systems(***kwargs*)
Find Systems using keyword arguments as search parameters

Parameters ****kwargs** – Search parameters

Returns List of Systems matching the search parameters

Return type List[*System*]

find_unique_request(***kwargs*)
Find a unique request

Note: If 'id' is a given keyword argument then all other parameters will be ignored.

Parameters ****kwargs** – Search parameters

Returns The Request if found, None otherwise

Return type *Request*, None

Raises FetchError – More than one matching Request was found

find_unique_system(***kwargs*)
Find a unique system

Note: If 'id' is a given keyword argument then all other parameters will be ignored.

Parameters ****kwargs** – Search parameters

Returns The System if found, None otherwise

Return type *System*, None

Raises FetchError – More than one matching System was found

get_config()
Get configuration

Returns Configuration dictionary

Return type dict

get_instance(*instance_id*)
Get an Instance

Parameters **instance_id** – The Id

Returns The Instance

get_instance_status(*instance_id*)
Get an Instance's status

Parameters **instance_id** – The Id

Returns The Instance's status

get_logging_config(*system_name=None*, *local=False*)
Get a logging configuration

Note that the system_name is not relevant and is only provided for backward-compatibility.

Parameters **system_name** (*str*) – UNUSED

Returns The configuration object

Return type dict

get_queues()
Retrieve all queue information

Returns List of all Queues

Return type List[*Queue*]

get_request(*request_id*)
Get a Request

Parameters **request_id** – The Id

Returns The Request

get_system(*system_id*, ***kwargs*)
Get a System

Parameters **system_id** – The Id

Returns The System

get_user(*user_identifier*)
Find a user

Parameters **user_identifier** (*str*) – User ID or username

Returns The User

Return type *Principal*

get_version(***kwargs*)
Get Bartender, Brew-view, and API version information

Parameters ****kwargs** – Extra parameters

Returns Response object with version information in the body

Return type dict

initialize_instance(*instance_id*, *runner_id=None*)
Start an Instance

Parameters

- **instance_id** (*str*) – The Instance ID
- **runner_id** (*str*) – The PluginRunner ID, if any

Returns The updated Instance

Return type *Instance*

instance_heartbeat(*instance_id*)
Send an Instance heartbeat

Parameters **instance_id** (*str*) – The Instance ID

Returns True if the heartbeat was successful

Return type bool

pause_job(*job_id*)
Pause a Job

Parameters **job_id** (*str*) – The Job ID

Returns The updated Job

Return type *Job*

publish_event(**args*, ***kwargs*)
Publish a new event

Parameters

- ***args** – If a positional argument is given it's assumed to be an Event and will be used
- ****kwargs** – Will be used to construct a new Event to publish if no Event is given in the positional arguments

Keyword Arguments **_publishers** (*Optional[List[str]]*) – List of publisher names. If given the Event will only be published to the specified publishers. Otherwise all publishers known to Beergarden will be used.

Returns True if the publish was successful

Return type bool

remove_instance(*instance_id*)
Remove an Instance

Parameters **instance_id** (*str*) – The Instance ID

Returns True if the remove was successful

Return type bool

remove_job(*job_id*)
Remove a unique Job

Parameters **job_id** (*str*) – The Job ID

Returns True if removal was successful

Return type bool

Raises DeleteError – Couldn't remove Job

remove_system(***kwargs*)
Remove a unique System

Parameters ****kwargs** – Search parameters

Returns True if removal was successful

Return type bool

Raises FetchError – Couldn't find a System matching given parameters

resume_job(*job_id*)
Resume a Job

Parameters **job_id** (*str*) – The Job ID

Returns The updated Job

Return type *Job*

update_instance(*instance_id*, ***kwargs*)
Update an Instance status

Parameters **instance_id** (*str*) – The Instance ID

Keyword Arguments

- **new_status** (*str*) – The new status
- **metadata** (*dict*) – Will be added to existing instance metadata

Returns The updated Instance

Return type *Instance*

update_instance_status(*instance_id*, *new_status*)
Update an Instance status

Parameters

- **instance_id** (*str*) – The Instance ID
- **new_status** (*str*) – The new status

Returns The updated Instance

Return type *Instance*

update_request(*request_id*, *status=None*, *output=None*, *error_class=None*)
Update a Request

Parameters

- **request_id** (*str*) – The Request ID
- **status** (*Optional[str]*) – New Request status
- **output** (*Optional[str]*) – New Request output
- **error_class** (*Optional[str]*) – New Request error class

Returns The updated response

Return type Response

update_system(*system_id*, *new_commands=None*, ***kwargs*)
Update a System

Parameters

- **system_id** (*str*) – The System ID
- **new_commands** (*Optional[List[Command]]*) – New System commands

Keyword Arguments

- **add_instance** (Instance) – An Instance to append
- **metadata** (*dict*) – New System metadata
- **description** (*str*) – New System description
- **display_name** (*str*) – New System display name
- **icon_name** (*str*) – New System icon name

Returns The updated system

Return type *System*

upload_file(*file_to_upload*, *desired_filename=None*, *file_params=None*)
Upload a given file to the Beer Garden server.

Parameters

- **file_to_upload** – Can either be an open file descriptor or a path.
- **desired_filename** – The desired filename, if none is provided it
- **use the basename of the file_to_upload** (*will*) –
- **file_params** – The metadata surrounding the file. Valid Keys: See brewtils File model

Returns A BG file ID.

who_am_i()
Find user using the current set of credentials

Returns The User

Return type *Principal*

class brewtils.**SystemClient**(***kwargs*)
Bases: object

High-level client for generating requests for a Beer-garden System.

SystemClient creation: This class is intended to be the main way to create Beer-garden requests. Create an instance with Beer-garden connection information and a system name:

```
client = SystemClient(
    system_name='example_system',
    system_namespace='default',
    bg_host="host",
    bg_port=2337,
)
```

Note: Passing an empty string as the system_namespace parameter will evalutate to the local garden's default namespace.

Pass additional keyword arguments for more granularity:

version_constraint: Allows specifying a particular system version. Can be a version literal ('1.0.0') or the special value 'latest.' Using 'latest' will allow the SystemClient to retry a request if it fails due to a missing system (see Creating Requests).

default_instance: The instance name to use when creating a request if no other instance name is specified. Since each request must be addressed to a specific instance this is a convenience to prevent needing to specify the instance for each request.

always_update: If True the SystemClient will always attempt to reload the system definition before making a request. This is useful to ensure Requests are always made against the latest version of the system. If not set the System definition will be loaded when making the first request and will only be reloaded if a Request fails.

Loading the System: The System definition is lazily loaded, so nothing happens until the first attempt to send a Request. At that point the SystemClient will query Beer-garden to get a system definition that matches the system_name and version_constraint. If no matching System can be found a FetchError will be raised. If always_update was set to True this will happen before making each request, not only the first.

Making a Request: The standard way to create and send requests is by calling object attributes:

```
request = client.example_command(param_1='example_param')
```

In the normal case this will block until the request completes. Request completion is determined by periodically polling Beer-garden to check the Request status. The time between polling requests starts at 0.5s and doubles each time the request has still not completed, up to max_delay. If a timeout was specified and the Request has not completed within that time a `ConnectionTimeoutError` will be raised.

It is also possible to create the SystemClient in non-blocking mode by specifying blocking=False. In this case the request creation will immediately return a Future and will spawn a separate thread to poll for Request completion. The max_concurrent parameter is used to control the maximum threads available for polling.

```
# Create a SystemClient with blocking=False
client = SystemClient(
    system_name='example_system',
    system_namespace='default',
    bg_host="localhost",
    bg_port=2337,
    blocking=False,
)

# Create and send 5 requests without waiting for request completion
futures = [client.example_command(param_1=number) for number in range(5)]

# Now wait on all requests to complete
concurrent.futures.wait(futures)
```

If the request creation process fails (e.g. the command failed validation) and version_constraint is 'latest' then the SystemClient will check to see if a newer version is available, and if so it will attempt to make the request on that version. This is so users of the SystemClient that don't necessarily care about the target system version don't need to be restarted every time the target system is updated.

It's also possible to control what happens when a Request results in an ERROR. If the `raise_on_error` parameter is set to False (the default) then Requests that are not successful simply result in a Request with a status of ERROR, and it is the plugin developer's responsibility to check for this case. However, if `raise_on_error` is set to True then this will result in a `RequestFailedError` being raised. This will happen regardless of the value of the `blocking` flag.

Tweaking Beer-garden Request Parameters: There are several parameters that control how beer-garden routes / processes a request. To denote these as intended for Beer-garden itself (rather than a parameter to be passed to the Plugin) prepend a leading underscore to the argument name.

Sending to another instance:

```
request = client.example_command(
    _instance_name="instance_2", param_1="example_param"
)
```

Request with a comment:

```
request = client.example_command(
    _comment="I'm a beer-garden comment!", param_1="example_param"
)
```

Without the leading underscore the arguments would be treated the same as "param_1" - another parameter to be passed to the plugin.

Request that raises:

```
client = SystemClient(
    system_name="foo",
    system_namespace='default',
    bg_host="localhost",
    bg_port=2337,
)

try:
    client.command_that_errors(_raise_on_error=True)
except RequestFailedError:
    print("I could have just ignored this")
```

Parameters

- **system_name** (*str*) – Name of the System to make Requests on
- **system_namespace** (*str*) – Namespace of the System to make Requests on
- **version_constraint** (*str*) – System version to make Requests on. Can be specific ('1.0.0') or 'latest'.
- **default_instance** (*str*) – Name of the Instance to make Requests on
- **always_update** (*bool*) – Whether to check if a newer version of the System exists before making each Request. Only relevant if `version_constraint='latest'`
- **timeout** (*int*) – Seconds to wait for a request to complete. 'None' means wait forever.
- **max_delay** (*int*) – Maximum number of seconds to wait between status checks for a created request
- **blocking** (*bool*) – Flag indicating whether creation will block until the Request is complete or return a Future that will complete when the Request does
- **max_concurrent** (*int*) – Maximum number of concurrent requests allowed. Only has an effect when blocking=False.
- **raise_on_error** (*bool*) – Flag controlling whether created Requests that complete with an ERROR state should raise an exception
- **bg_host** (*str*) – Beer-garden hostname
- **bg_port** (*int*) – Beer-garden port
- **bg_url_prefix** (*str*) – URL path that will be used as a prefix when communicating with Beer-garden. Useful if Beer-garden is running on a URL other than '/'.
- **ssl_enabled** (*bool*) – Whether to use SSL for Beer-garden communication
- **ca_cert** (*str*) – Path to certificate file containing the certificate of the authority that issued the Beer-garden server certificate
- **ca_verify** (*bool*) – Whether to verify Beer-garden server certificate
- **client_cert** (*str*) – Path to client certificate to use when communicating with Beer-garden
- **api_version** (*int*) – Beer-garden API version to use
- **client_timeout** (*int*) – Max time to wait for Beer-garden server response
- **username** (*str*) – Username for Beer-garden authentication
- **password** (*str*) – Password for Beer-garden authentication
- **access_token** (*str*) – Access token for Beer-garden authentication
- **refresh_token** (*str*) – Refresh token for Beer-garden authentication

create_bg_request(*command_name*, ***kwargs*)

Create a callable that will execute a Beer-garden request when called.

Normally you interact with the SystemClient by accessing attributes, but there could be certain cases where you want to create a request without sending it.

Example:

```
client = SystemClient(host, port, 'system', blocking=False)

# Create two callables - one with a parameter and one without
uncreated_requests = [
    client.create_bg_request('command_1', arg_1='Hi!'),
    client.create_bg_request('command_2'),
]

# Calling creates and sends the request
# The result of each is a future because blocking=False on the SystemClient
futures = [req() for req in uncreated_requests]

# Wait for all the futures to complete
concurrent.futures.wait(futures)
```

Parameters

- **command_name** (*str*) – Name of the Command to send
- **kwargs** (*dict*) – Will be passed as parameters when creating the Request

Returns Partial that will create and execute a Beer-garden request when called

Raises `AttributeError` – System does not have a Command with the given command_name

load_bg_system()

Query beer-garden for a System definition

This method will make the query to beer-garden for a System matching the name and version constraints specified during SystemClient instance creation.

If this method completes successfully the SystemClient will be ready to create and send Requests.

Returns None

Raises `FetchError` – Unable to find a matching System

send_bg_request(**args*, ***kwargs*)

Actually create a Request and send it to Beer-garden

Note: This method is intended for advanced use only, mainly cases where you're using the SystemClient without a predefined System. It assumes that everything needed to construct the request is being passed in `kwargs`. If this doesn't sound like what you want you should check out `create_bg_request`.

Parameters

- **args** (*list*) – Unused. Passing positional parameters indicates a bug
- **kwargs** (*dict*) – All necessary request parameters, including Beer-garden internal parameters

Returns A completed Request object blocking=False: A future that will be completed when the Request does

Return type blocking=True

Raises `ValidationError` – Request creation failed validation on the server

brewtils.**get_easy_client**(***kwargs*)
Easy way to get an EasyClient

The benefit to this method over creating an EasyClient directly is that this method will also search the environment for parameters. Kwargs passed to this method will take priority, however.

Parameters ****kwargs** – Options for configuring the EasyClient

Returns The configured client

Return type *brewtils.rest.easy_client.EasyClient*

brewtils.**get_argument_parser**()
Get an ArgumentParser pre-populated with Brewtils arguments

This is helpful if you're expecting additional command line arguments to a plugin startup script.

This enables doing something like:

```
def main():
    parser = get_argument_parser()
    parser.add_argument('positional_arg')

    parsed_args = parser.parse_args(sys.argv[1:])

    # Now you can use the extra argument
    client = MyClient(parsed_args.positional_arg)

    # But you'll need to be careful when using the 'normal' Brewtils
    # configuration loading methods:

    # Option 1: Tell Brewtils about your customized parser
    connection = get_connection_info(cli_args=sys.argv[1:],
                                     argument_parser=parser)

    # Option 2: Use the parsed CLI as a dictionary
    connection = get_connection_info(**vars(parsed_args))

    # Now specify connection kwargs like normal
    plugin = RemotePlugin(client, name=...
                          **connection)
```

IMPORTANT: Note that in both cases the returned `connection` object **will not** contain your new value. Both options just prevent normal CLI parsing from failing on the unknown argument.

Returns ArgumentParser: Argument parser with Brewtils arguments loaded

brewtils.**get_connection_info**(*cli_args=None*, *argument_parser=None*, ***kwargs*)
Wrapper around `load_config` that returns only connection parameters

Parameters

- **cli_args** (*list, optional*) – List of command line arguments for configuration loading
- **argument_parser** (*ArgumentParser, optional*) – Argument parser to use when parsing cli_args. Supplying this allows adding additional arguments prior to loading the configuration. This can be useful if your startup script takes additional arguments.
- ****kwargs** – Additional configuration overrides

Returns dict: Parameters needed to make a connection to Beergarden

brewtils.**load_config**(*cli_args=True*, *environment=True*, *argument_parser=None*, *bootstrap=False*, ***kwargs*)

Load configuration using Yapconf

Configuration will be loaded from these sources, with earlier sources having higher priority:

1. `**kwargs` passed to this method
2. Command line arguments (if `cli_args` argument is not False)
3. **Environment variables using the `BG_` prefix (if `environment` argument** is not False)
4. Default values in the brewtils specification

Parameters

- **cli_args** (*Union[bool, list], optional*) – Specifies whether command line should be used as a configuration source - True: Argparse will use the standard sys.argv[1:] - False: Command line arguments will be ignored when loading configuration - List of strings: Will be parsed as CLI args (instead of using sys.argv)
- **environment** (*bool*) – Specifies whether environment variables (with the `BG_` prefix) should be used when loading configuration
- **argument_parser** (*ArgumentParser, optional, deprecated*) – Argument parser to use when parsing cli_args. Supplying this allows adding additional arguments prior to loading the configuration. This can be useful if your startup script takes additional arguments. See get_argument_parser for additional information.
- ****kwargs** – Additional configuration overrides

Returns The resolved configuration object

Return type `box.Box`

brewtils.**configure_logging**(*raw_config*, *namespace=None*, *system_name=None*, *system_version=None*, *instance_name=None*)

Load and enable a logging configuration from Beergarden

WARNING: This method will modify the current logging configuration.

The configuration will be template substituted using the keyword arguments passed to this function. For example, a handler like this:

```
handlers:
    file:
        backupCount: 5
        class: "logging.handlers.RotatingFileHandler"
        encoding: utf8
        formatter: default
        level: INFO
        maxBytes: 10485760
        filename: "$system_name.log"
```

Will result in logging to a file with the same name as the given system_name.

This will also ensure that directories exist for any file-based handlers. Default behavior for the Python logging module is to not create directories that do not already exist, which would dramatically lower the utility of templating.

Parameters

- **raw_config** – Configuration to apply

- **namespace** – Used for configuration templating
- **system_name** – Used for configuration templating
- **system_version** – Used for configuration templating
- **instance_name** – Used for configuration templating

Returns None

brewtils.**normalize_url_prefix**(*url_prefix*)

Enforce a consistent URL representation

The normalized prefix will begin and end with '/'. If there is no prefix the normalized form will be '/'.

Examples

INPUT NORMALIZED None '/' '' '/' '/' '/' 'example' '/example/' '/example' '/example/' 'example/' '/example/' '/example/' '/example/'

Parameters **url_prefix** (*str*) – The prefix

Returns The normalized prefix

Return type str

CHAPTER 6

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/beer-garden/brewtils/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

6.1.4 Write Documentation

Brewtils could always use more documentation, whether as part of the official Brewtils docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/beer-garden/brewtils/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up `brewtils` for local development.

1. Fork the `brewtils` repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/brewtils.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv brewtils
$ cd brewtils/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 brewtils test
$ nosetests
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.5, and 3.6. Check https://travis-ci.org/beer-garden/brewtils/pull_requests and make sure that the tests pass for all supported Python versions.

6.4 Tips

To run a subset of tests:

```
$ nosetests test/models_test.py:SystemTest.test_instance_names
```

CHAPTER 7

Credits

7.1 Development Leads

- Logan Asher Jones <loganasherjones@gmail.com>
- Matt Patrick

7.2 Contributors

None yet. Why not be the first?

CHAPTER 8

Brewtils Changelog

8.1 3.0.0

Date: 11/10/20

Note: This is a major upgrade with several breaking changes. Please see the Upgrade Guide for all changes.

8.1.1 New Features

- Plugins now automatically load configuration from CLI and environment variables
- Logging configuration is loaded automatically when Plugins are created
- No longer need to pass connection information to System/Easy/Rest Clients
- Parameter choices definition can be a non-list iterable (beer-garden/#512)
- It's now easier to specify an alternate parent when making a request (beer-garden/#336)
- SchemaParser can now directly serialize dicts and Boxes (#239)

8.1.2 Bug Fixes

- EasyClient.get_instance_status is deprecated but now actually returns the instance status

8.1.3 Other Changes

- Plugins are now multi-threaded by default (#47)
- Better error messages when using SystemClient with raise_on_error=True (beer-garden/#689)
- Various deprecated names have been removed
- Can now defer setting a Plugin client

- EasyClient.get_version returns actual version information instead of Response object
- Using a pika version <1 is deprecated

8.2 2.4.15

Date: 10/13/20

8.2.1 Bug Fixes

- Fixing command invocation error when request has no parameters (beer-garden/#351)

8.3 2.4.14

Date: 1/30/20

8.3.1 Bug Fixes

- Better error handling if a request exceeds 16MB size limit (beer-garden/#308)

8.4 2.4.13

Date: 1/13/20

8.4.1 Bug Fixes

- Requests republished to rabbit are now persistent (beer-garden/#397)

8.5 2.4.12

Date: 1/10/20

8.5.1 Other Changes

- Reverting a log message level that was incorrectly set to INFO

8.6 2.4.11

Date: 12/9/19

8.6.1 Other Changes

- Plugins always attempt to notify Beer-garden when terminating (beer-garden/#376)

8.7 2.4.10

Date: 11/12/19

8.7.1 Bug Fixes

- Plugins can now survive a rabbitmq broker restart (beer-garden/#353, beer-garden/#359)

8.8 2.4.9

Date: 10/30/19

8.8.1 Bug Fixes

- Fixed issue with callbacks in RequestConsumer when using Pika v1 (beer-garden/#328)

8.9 2.4.8

Date: 9/5/19

8.9.1 New Features

- Better control over how specific error types are logged (beer-garden/#285)

8.9.2 Bug Fixes

- Decorators now work with non-JSON resources loaded from a URL (beer-garden/#310)

8.10 2.4.7

Date: 6/27/19

8.10.1 New Features

- Can now specify a name and version in the `system` decorator (beer-garden/#290)

8.10.2 Bug Fixes

- SystemClient now correctly handles versions with suffixes (beer-garden/#283)

8.10.3 Other Changes

- Added compatability with Pika v1 (#130)

8.11 2.4.6

Date: 4/19/19

8.11.1 Bug Fixes

- Using new pika heartbeat instead of heartbeat_interval (#118)
- @parameters now accepts any iterable, not just lists (beer-garden/#237)

8.11.2 Other Changes

- Support for new header-style authentication token (#122)
- Added EasyClient.get_instance, deprecated get_instance_status (beer-garden/#231)
- Parameters with is_kwarg on command without **kwargs will raise (beer-garden/#216)

8.12 2.4.5

Date: 2/14/19

8.12.1 Bug Fixes

- Fixed a warning occuring with newer versions of Marshmallow (#111)

8.12.2 Other Changes

- Adding EasyClient to __all__ (beer-garden/#233)

8.13 2.4.4

Date: 1/7/19

8.13.1 Bug Fixes

- RabbitMQ connections now deal with blocked connections (beer-garden/#203)
- Plugin will use url_prefix kwarg if bg_url_prefix not given (beer-garden/#186)
- Always respecting parameter choices definition changes (beer-garden/#58)

8.14 2.4.3

Date: 11/16/18

8.14.1 New Features

- Added instance retrieve and delete methods to clients (#91)

8.14.2 Bug Fixes

- Logging API now respects all connection parameters (#94)

8.15 2.4.2

Date: 10/7/18

8.15.1 New Features

- Ability to specify a timeout for Beergarden communication (beer-garden/#87)
- `parameters` decorator for cleaner command definitions (beer-garden/#82)

8.15.2 Bug Fixes

- Fixed error when republishing a message to RabbitMQ (beer-garden/#88)

8.16 2.4.1

Date: 09/11/18

8.16.1 Other Changes

- Changed Plugin warning type so it won't be displayed by default

8.17 2.4.0

Date: 09/5/18

8.17.1 New Features

- Added job scheduling capability (beer-garden/#10)
- Added support for authentication / users (beer-garden/#35)
- Plugins will load log level from the environment (bartender/#4)
- RestClient now exposes `base_url` (#58)
- SystemClient can wait for a request to complete instead of polling (#54)
- Allowing custom argument parser when loading configuration (#67)
- Support for TLS connections to RabbitMQ (#74)
- Warning for future change to plugin max_concurrent default value (#79)
- Added methods `get_config` to RestClient, `can_connect` to EasyClient

8.17.2 Other Changes

- Renamed PluginBase to Plugin (old name is aliased)

8.18 2.3.7

Date: 07/11/18

8.18.1 New Features

- Current request can be accessed using `self._current_request` (beer-garden/#78)

8.18.2 Bug Fixes

- Updating import problem from lark-parser #61
- Pinning setup.py versions to prevent future breaks

8.19 2.3.6

Date: 06/06/18

8.19.1 Other Changes

- Added *has_parent* to request model

8.20 2.3.5

Date: 4/17/18

8.20.1 Bug Fixes

- Using *simplejson* package to fix JSON parsing issue in Python 3.4 & 3.5 (#48, #49)

8.21 2.3.4

Date: 4/5/18

8.21.1 New Features

- Python 3.4 is now supported (#43)
- Now using Yapconf for configuration parsing (#34)
- Parameter types can now be specified as native Python types (#29)
- Added flag to raise an exception if a request created with `SystemClient` completes with an 'ERROR' status (#28)

8.21.2 Other Changes

- All exceptions now inherit from `BrewtilsException` (#45)
- Removed references to `Brewmaster` exception classes (#44)
- Requests with JSON `command_type` are smarter about formatting exceptions (#27)
- Decorators, `RemotePlugin`, and `SystemClient` can now be imported directly from the `brewtils` package

8.22 2.3.3

Date: 3/20/18

8.22.1 Bug Fixes

- Fixed bug where request updating could retry forever (#39)

8.23 2.3.2

Date: 3/7/18

8.23.1 Bug Fixes

- Fixed issue with multi-instance remote plugins failing to initialize (#35)

8.24 2.3.1

Date: 2/22/18

8.24.1 New Features

- Added `description` keyword argument to `@command` decorator

8.25 2.3.0

Date: 1/26/18

8.25.1 New Features

- Added methods for interacting with the Queue API to RestClient and EasyClient
- Clients and Plugins can now be configured to skip server certificate verification when making HTTPS requests
- Timestamps now have true millisecond precision on platforms that support it
- Added `form_input_type` to Parameter model
- Plugins can now be stopped correctly by calling their `_stop` method
- Added Event model

8.25.2 Bug Fixes

- Plugins now additionally look for `ca_cert` and `client_cert` in `BG_CA_CERT` and `BG_CLIENT_CERT`

8.25.3 Other Changes

- Better data integrity by only allowing certain Request status transitions

8.26 2.2.1

Date: 1/11/18

8.26.1 Bug Fixes

- Nested requests that reference a different beer-garden no longer fail

8.27 2.2.0

Date: 10/23/17

8.27.1 New Features

- Command descriptions can now be changed without updating the System version
- Standardized Remote Plugin logging configuration
- Added domain-specific language for dynamic choices configuration
- Added `metadata` field to Instance model

8.27.2 Bug Fixes

- Removed some default values from model `__init__` functions
- System descriptors (description, display name, icon name, metadata) now always updated during startup
- Requests with output type 'JSON' will now have JSON error messages

8.27.3 Other changes

- Added license file

8.28 2.1.1

Date: 8/25/17

8.28.1 New Features

- Added `updated_at` field to `Request` model
- `SystemClient` now allows specifying a `client_cert`
- `RestClient` now reuses the same session for subsequent connections
- `SystemClient` can now make non-blocking requests
- `RestClient` and `EasyClient` now support PATCHing a `System`

8.28.2 Deprecations / Removals

- `multithreaded` argument to `PluginBase` has been superseded by `max_concurrent`
- These decorators are now deprecated - `@command_registrar`, instead use `@system` - `@plugin_param`, instead use `@parameter` - `@register`, instead use `@command`
- These classes are now deprecated - `BrewmasterSchemaParser`, instead use `SchemaParser` - `BrewmasterRestClient`, instead use `RestClient` - `BrewmasterEasyClient`, instead use `EasyClient` - `BrewmasterSystemClient`, instead use `SystemClient`

8.28.3 Bug Fixes

- Reworked message processing to remove the possibility of a failed request being stuck in `IN_PROGRESS`
- Correctly handle custom form definitions with a top-level array
- Smarter reconnect logic when the RabbitMQ connection fails

8.28.4 Other changes

- Removed dependency on `pyopenssl` so there's need to compile any Python extensions
- Request processing now occurs inside of a `ThreadPoolExecutor` thread
- Better serialization handling for epoch fields

Python Module Index

b

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W